University of Zurich[UZH]

Department of Informatics

Measuring Abstraction with Footprinting

Confronting Software Models with their Purposes

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology of the University of Zurich

for the degree of
Doctor of Science

by
Cédric Jeanneret
from Travers NE, Switzerland

Accepted on the recommendation of
Prof. Dr. Martin Glinz
Prof. Robert B. France, PhD

2013

University of Zurich[UZH]

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, February 2013

Head of the PhD committee for informatics:
Prof. Abraham Bernstein, PhD

Everything should be made
as simple as possible, but no simpler.

—Albert Einstein

(Physicist, 1879-1955)

Not everything that can be counted counts,
and not everything that counts can be counted.

—William Bruce Cameron

(Sociologist)

The solutions all are simple... after you have arrived at them.
But they're simple only when you know already what they are.

—Robert M. Pirsig

(Philosopher, 1928)

Abstract

A model is an abstract representation of an entity created for a given purpose. Software engineers use many kinds of models in various contexts, ranging from informal models sketched on a whiteboard to executable models deployed in a production environment. Among other qualities, a "good" model is at the right level of abstraction. Indeed, a model that is too abstract may lead to imprecise or incorrect conclusions while a model that is too detailed is larger and more complex than necessary. One way or the other, a model loses much of its value if it is at the wrong level of abstraction.

Today, modelers must rely on their instinct and experience to decide how much and what detail is worth including in a model. This ad-hoc form of modeling may result in models that are either too abstract or too detailed for their intended use. We believe that building models at the right level of abstraction is too important to depend solely on the modeler's skill. Thus, our research aims at providing modelers with

an objective measurement of a model's abstractness and systematic guidance to attain the right level of abstraction.

In this thesis, we present MiRiA (which stands for Modeling at the Right level of Abstraction), an approach split in two parts. The first part is a metamodeling activity, where the purpose of a model is captured and operationalized with a set of model operations — such as analysis, queries, simulations and transformations — and a metamodel supporting them. This step ensures that both the modeler and the users share the same understanding of the model's purpose.

The second part of MiRiA consists in confronting a model with its usage by a set of model operations. When performed on a model, an operation computes new information based on the information stored in the model. During its execution, it navigates through the content of the model and gathers relevant information from model elements. The set of elements accessed by a model operation during its execution forms the footprint of that operation. The comparison between the actual and expected footprint reveals excessive information in the model or suggests information missing from it. Based on these indications, the modelers can improve their models by simplifying or extending them until they reach the right level of abstraction.

Contents

List of Figures

List of Tables

Chapter 1

Synopsis

Software engineering is essentially a model-based problem solving activity, because it involves the analysis of a problem, the conception of a solution and the expression of this solution in a high-level programming language [FR07]. A *model* is an abstract representation of an entity for a given purpose. In software engineering, many kinds of models are used in various contexts, ranging from informal models sketched on a whiteboard to executable models deployed in a production environment. Models are used for various purposes, including comprehension or manipulation of the entity, communication, planning or prediction [Rot89].

In our work, we are especially interested in models used for deriving new information about a system or a process from the information present in the model. We call such an inference a *model operation*. Typical model operations include analyses, measurements, simulations,

transformations to other models and model queries. For example, Cortelessa *et al.* propose to transform UML models to queueing network for performance analysis [CM00] while Cheung *et al.* presented in [CRMG08] an approach to estimate the reliability of a software system based on models available early in the development of the system. To better leverage modeling techniques in the development of software systems, *Model Driven Engineering* (MDE) approaches advocate the creation of abstract models of software systems and their systematic transformation to concrete implementations [Sel03]. These model operations are increasingly automated to improve both the productivity of engineers and the reliability of the operation.

The strength of models is related to the ideas of abstraction and separation of concerns: one can focus on important aspects of a software system by ignoring its irrelevant details. Still, what is considered important and what can be abstracted depends on the purpose of the model [GJM02]. Failing to achieve a proper separation of concerns or to attain the right level of abstraction can have dramatic consequences on meeting the purpose. If a model lacks some important details for a given purpose, one may draw imprecise or incorrect conclusions from the model. On the opposite, if a model contains too much detail for its purpose, its comprehensibility may suffer pointlessly [Nug09]. Furthermore, such a model will be larger than necessary, requiring therefore more time for its creation than necessary. In other words, a model loses much of its value if it is not at the right level of abstraction for its purpose.

Today, deciding how much and which details to mention in a model is based on the modelers' skill and experience. As Kramer observes in [Kra07], not everyone is equally endowed for abstract thinking. Improving the education of abstraction skills can only partially solve the problem. We think that building models at the right level of abstraction is too important to depend solely on the modeler's skills. Albeit much research effort has been devoted to modeling and abstraction, the state of the art cannot support modelers in objectively assessing and systematically improving the level of abstraction of their models in a satisfactory manner.

1.1 State of the Art

In this section, we depict the state of the art in modeling software systems and assessing the quality of models. It is organized as follows: First, we briefly review theories of modeling for software systems. Since our work is about abstraction and separation of concerns in general, we introduce viewpoint frameworks and techniques to extract views from a model. Then we present the current understanding of abstraction in software engineering. Finally, we review the various proposals made to assess and evaluate the quality of models.

1.1.1 Theory of Modeling

The importance of models in software engineering led many researchers to investigate a theory of modeling for software systems,

describing what models are and how they are built. Contributions to such a theory of modeling were made by Ludewig [Lud03], Bézivin [Béz05], Kühne [Küh06], Müller *et al.* [MFBC12], Selic [Sel03] and Seidewitz [Sei03]. Many of these articles build on Stachowiak's general model theory [Sta73] which defines three criteria to distinguish models from other artifacts: (a) a model is related to an entity, which may not exist yet (mapping criterion), (b) a model only reflects a selection of the entity's properties (reduction criterion) and (c) a model can replace the entity for some purpose (pragmatic criterion).

Most of these papers consider models as artifacts and ignore their creation. On the opposite, Hoppenbrouwers *et al.* propose a theory of the modeling process, accounting for subjective aspects of modeling [HPvdW05]. When modeling, participants (such as domain experts and system analysts) attempt to reconcile representations of their conception of the world. The model (as a product) is the minutes resulting from these modeling dialogues.

While models can be physical (such as architectural models), most models in software engineering are conceptual, thus, they are expressed in a modeling language (*e.g.*, UML [OMG11c] or Adora [GBJ02]). If the model is to be processed by a tool, it must be expressed in a modeling language with at least a formal syntax and, if available, precisely defined semantics [HR04]. The syntax of models is usually defined with a metamodel, which is itself a model expressed in a metamodeling language (*e.g.*, MOF [OMG11a]). While van Gigch defines a metamodel as a model of the modeling process in

its entirety (that is, it defines requirements on the modeling activity) [vG91], many authors of metamodeling papers (such as Kleppe [Kle08] or Kühne [Küh06]) focus on the abstract syntax model of languages (that is, the definition of modeling concepts and their relationships). Kermeta [MFJ05] is a notable exception, as it allows a metamodeler to define operational semantics as behaviors defined in the language metamodel.

Only few papers focus on the purposes of a model. In a literature review of modeling theories, Muller *et al.* propose a notation to document the representation relationships among things involved in software development [MFBC12]. The intention of a model is at the heart of their notation. However, they consider intentions as sets and represent them as Venn diagrams in their notations. Thus, this notation allows visualizing intersecting or overlapping intentions, but not their internal content.

1.1.2 Viewpoints and Views Extraction

A single model should not represent all parts, aspects and details of a desired software system. Such a model would be very large and difficult to grasp. Furthermore, stakeholders have different views on the system to be developed, especially during the elicitation of requirements for a software system. Thus, a system is often seen from multiple viewpoints, whose resulting views are partial descriptions of the system. For Nuseibeh *et al.* [NKF94], a viewpoint consists of

(1) a style (the modeling language and its notation), (2) a work plan describing the development process of the viewpoint including possible consistency checking or construction operations, (3) a domain defining the area of concern with respect to the overall system, (4) a specification describing the viewpoint's domain using the viewpoint's style (in other words, the view of the system from the viewpoint) and (5) a work record keeping track of the development history within the viewpoint. A software development method can be defined as a set of related viewpoints.

Similarly, software architectures are described with various separate, but interrelated views. In his seminal article [Kru95], Kruchten defined four standard views: Logical, Process, Physical and Development. A fifth one, Scenarios, is used to show that elements from the four views work together seamlessly. According to the IEEE 1471 standard [IEE00], a viewpoint captures the conventions for constructing, interpreting and analyzing a particular kind of view. Thus, a viewpoint defines — among others — a modeling language, model operations that can be applied to views and stakeholders whose concerns are addressed in the views. The standard does not propose nor impose any viewpoint, as the selection of viewpoints depends on the project. Still, many books like [CGB+02] or [RW05] propose a catalogue of viewpoints beside those identified by Kruchten in [Kru95]. However, they do not offer guidelines to define viewpoints nor techniques to validate them.

While much research effort was spent on techniques to verify the consistency among viewpoints and to merge viewpoints into a single one [Jea08], these issues are still considered as challenges in the MDE community [FR07]. Adora [GBJ02] and the orthographic modeling environment [AS08] address this problem from the opposite direction by generating views from a single model. Adora is a modeling language in which the various facets of a system — such as its structure, its behavior and its context — are modeled in a single hierarchical model. Because these facets are modeled in a single model, there cannot be any inconsistencies among them and the modelers do not have to refer concurrently to multiple diagrams, in contrary to languages where these facets are modeled separately (*e.g.*, UML). For editing and navigating Adora models, the editor provides modelers with two abstraction mechanisms: vertical and horizontal abstraction. Vertical abstraction hides the content of a node (Adora models are hierarchical). For example, the modelers can visualize a given sub-system as a white or a black box, that is, the tool can show or hide its internal components, its behavior, its functionalities and the scenarios it is involved in. Horizontal abstraction hides all elements related to a particular facet. For example, modelers can visualize the structure and the context of a system without displaying its behavior. The orthographic modeling environment [AS08] pushes the idea of generating views from a single model to its extreme: In this environment, all software artefacts such as code and design models are generated views.

Extracting a view from a model is a form of model slicing, where some elements of a model are extracted based on a slicing criterion. For example, Korel *et al.* have presented in [KSTV03] an approach that extracts the parts of the model that affect an element of interest from an UML state machine. Metamodel pruning [SMBJ09] consists of slicing a large metamodel to create a smaller metamodel with the constructs that are of interest. Recently, Blouin *et al.* created Kompren [BCBB12], a domain specific language for model slicers. These techniques only partially solve our problem: They provide modelers with tools to extract views from a large model, but the modelers must still decide what is relevant for the purpose at hand.

1.1.3 Abstraction

Abstraction is a word with many meanings. In software engineering, the notion of abstraction is bound to both the generic nature of concepts and the simplicity of their representations. Liskov and Guttag defined it in [LG00] as follows:

> The process of abstraction can be seen as an application of a many-to-one mapping. It allows us to forget information and consequently to treat things that are different as if they were the same. We do this in the hope of simplifying our analysis by separating attributes that are relevant from those that are not.

Ludewig gives two striking examples to illustrate the benefits of abstraction [Lud03]:

> They who note that the change from high tide to low tide and from low tide to high tide follows a certain rhythm can prepare for, or make use of it. Those who learn that a certain class of animals rather than one single living creature is fast, strong, and dangerous, have improved their chances for survival.

However, abstraction is not free of danger. Problems arise when abstractions are used for purposes they were not made for. Kramer illustrates the problem with a traveler misinterpreting the London tube map as a geographic map in London [Kra07]. Liskov recalls that in mathematics, $(\frac{8}{3}) \times 3$ and $5 + 3$ are both abstracted to the concept represented by the numeral 8, while this abstraction is problematic on many computing machines [LG00].

In software engineering, the process of abstraction plays a central role in software reuse. In his definitive survey [Kru92], Krüger observes that abstractions have a fixed part, a flexible part (the parameters) and a hidden part (its implementation). Once an abstraction has been selected, it can be reused, by first specializing it and then integrating it into the system.

Many abstraction mechanisms have been identified in the literature:

Aggregation subsumes several components under a composite, ignoring the distinction between them [MMP88].

Classification subsumes all instances sharing certain selected property under one class, so that they can be treated uniformly [MMP88].

Generalization subsumes all subclasses defined by a common subset of properties under one super class, increasing the scope of uniform treatment [MMP88].

Parameterization abstracts from the identity of data by replacing it with parameters, generalizing a module so that it can be used in more situations [LG00].

Specification abstracts from implementation details to the behavior users can depend on, isolating modules from another's implementations [LG00].

The abstraction process creates an "abstraction of" relation among concepts. This relation is antisymmetric and transitive (except for classification). Thus, the relation defines a partial ordering on concepts, creating a hierarchy of abstraction and making possible to consider a concept at different levels of abstraction.

When developing software, the opposite of abstraction is concretization or refinement [Wir83]. Theoretically, there is no limit at the top or at the bottom of the hierarchy. Because software is ultimately run on hardware, the ground level is often defined as machine code or the level at which a concept can be automatically translated to

machine code. Thus, one possible way to measure the level of abstraction of a concept is the amount of detail required to represent or execute this concept in terms of hardware [Kle08]. Inspired by the work of Shannon in information theory, Salzer proposes to count the number of possible implementations of a requirement to quantify the implementation information contained in it [Sal10]. However, these measures assume that the main purpose of a concept is to be executed.

Some software engineering methods are built around a fixed number of levels in this hierarchy of abstraction. For example, the MDA architecture [OMG03] prescribes the modeling of a system at three predefined levels of abstraction: the computational independent model (CIM), the platform independent model (PIM) and the platform specific model (PSM). In software product management, the Requirements Abstraction Model (RAM) defines four levels of abstraction in which incoming requirements are classified and worked-up [GW05]. These levels are:

Product: a requirement at this level can be compared with product strategies

Feature: a requirement at this level describes a feature, abstractly

Function: a requirement at this level describes functions in a testable and unambiguous manner

Component: a requirement at this level provides examples of or limitations on the implementations

These predefined levels of abstraction (three for MDA, four for RAM) form an ordinal scale for the measurement of abstraction, where each level corresponds to models (or requirements) fit for a predefined purpose. In these approaches, the evaluation is subjective. Thus, the reliability of this evaluation depends on the experience of the modeler or the product manager, respectively.

Since abstracting from a concept produces a simpler concept, the level of abstraction of a concept can be assessed with the complexity of its representation. However, this angle of attack reduces a characteristic that is difficult to measure and to grasp to another characteristic that is as elusive as abstraction. A preliminary study of Saitta and Zucker showed that complexity may either increase or decrease with abstraction depending on their definitions, suggesting that complexity and abstraction may not be monotonically coupled to each other [SZ07].

1.1.4 Model Quality

Based on a literature survey, Davis *et al.* proposed a list of 24 desirable properties of software requirements specifications (SRS) [DOJ+93]. Among others, a good SRS is at the right level of detail. While the authors provide measures for the majority of the properties they propose, they concede that this property cannot be measured because it is highly scenario-dependant.

In [LSS94], Lindland *et al.* proposed the first systematic approach to identifying model quality goals and means to achieve them. In this framework based on semiotics (the study of symbols), models are seen as a set of statements. Models are compared to three other sets of statements: The *language* is the set of statements expressible in the modeling language. The *domain* is the set of all possible statements that would be correct and relevant while the *audience interpretation* is the set of statements that the audience thinks a model contains. Their framework defines three types of model quality. Syntactic quality is how well a model corresponds to the language. Semantic quality is how well the model corresponds to the domain. Finally, pragmatic quality is how well a model corresponds to the audience interpretation.

The framework has been extended by Krogstie *et al.* [KSJ06], who added several quality goals based on the levels of Stamper's semiotic ladder, such as social quality (having the goal of feasible agreement) and empirical quality (comprising comprehensibility matters such as layout for graphs and readability indexes for text).

However, it is difficult to make reliable and reproducible evaluations of models by using this framework. While syntactic quality can be measured objectively, the domain and stakeholder's mind cannot be inspected formally. Thus, many qualities must be evaluated subjectively. This issue does not only concern their framework: in a literature survey [Moo05], Moody identified the lack of objective measurement as one of the issues in model quality research.

Schuette *et al.* assume that the subjectivity of the model is an important factor in model quality. To keep this subjectivity under control, they proposed guidelines based on the generally accepted accounting principles [SR98]. Two guidelines are particularly relevant to our research: the principle of construct adequacy and the principle of clarity. The adequacy of a model implies the consideration, which specific information objects need to be included in the model. This raises the problem of getting a consensus among model builders and model users. The objective of clarity subsumes the filtering of information to meet the need of the model users. Thus, model users have to decide which information they need.

In another literature survey, Mohageghi *et al.* identified six model quality goals [MDN09]:

Correctness: a model only includes valid statements and adheres to the language syntax
Completeness: a model contains all the information that is relevant
Consistency: a model does not contain contradictions
Comprehensibility: the model is understandable for human users and tools
Confinement: a model does not have unnecessary information
Changeability: a model can be changed or evolved rapidly

The presence of an element in a model either improves the model's completeness if it is relevant or lowers the model's confinement if it is not. Mohageghi *et al.* identify several modeling guidelines aimed

at improving, among others, these two characteristics. Some of these guidelines were proposed by Berenbach in [Ber04]. For example, every concrete use case in UML must be defined with one or more sequence, collaboration, activity or state diagrams. In his case, this heuristic indeed improves the completeness of the model, because he uses models to extract requirements that completely describe the black-box behavior of a system. But if a stakeholder is only interested in the overview of the functionality of a system, these additional diagrams impede the confinement of a model. Similarly, most guidelines proposed in [Amb05] are independent of the purpose of the model. Furthermore, these guidelines only apply for UML models.

The quality of a model also depends on its modeling language. If models are to represent phenomena of the real-world, then the modeling language must be expressive enough so that these phenomena can be captured. Ontological analysis consists of mapping the constructs of a modeling language on the constructs of an ontological model [WW93]. This analysis evaluates the ontological clarity and expressiveness of modeling languages and can reveal anomalies such as (i) excessive constructs, (ii) overloaded constructs, (iii) redundant constructs and (iv) construct deficiencies. A linguistic construct is excessive if there is no corresponding ontological construct. In contrast, a linguistic construct is overloaded if it is mapped to two or more ontological constructs. Two (or more) linguistic constructs are redundant if they are mapped to the same ontological construct. Fi-

nally, the language has a construct deficiency if there is an ontological construct without a corresponding linguistic construct.

The pertinence of the analysis strongly depends on the ontological model chosen for the comparison. A well known ontological model is the one that Wand and Weber created based on an ontology defined by Bunge (this model is named Bunge-Wand-Weber model or BWW) [WW93]. It covers most phenomena relevant to software systems, such as thing, class, state, transformation and system. Still, when comparing the Entity-Relationship notation with the BWW models, the analysis reports many defects that are actually not defects, such as construct deficiencies for states and transformations. Thus, the problem of deciding which construct is relevant for a purpose remains unsolved [GR04].

1.2 Motivation

Software engineering is, in general, a model-based problem solving activity [FR07] and thus involves many models [Lud03]. The strength of modeling stems from the idea of abstraction, which is a form of separation of concerns. Thus, models lose much of their value if they are not at the right level of abstraction for their purpose. This motivates us to investigate objective measurement of a model's abstractness. However, such a measurement only allows detecting models at the wrong level of abstraction. Thus, we also need to provide systematic

guidance to improve the level of abstraction of such models. With the increasing use of MDE approaches in practice, the significance of such contributions will grow steadily in the future.

As we have seen in the previous section, this problem is beyond the current state of the art. Much work has been done in explaining what models are and how they are built, but little research has been done to describe why they are built. While viewpoints define the various views (and their relationships) in a software specification or in an architecture description, they do not offer guidance to define new viewpoints, nor methods to validate that a view actually satisfies the needs of the stakeholders using it. Abstraction measures are either independent of the model's purpose or only defined for some predefined purposes. Furthermore, most of these measures are subjective and most objective measures are not computable. Finally, these measures are mostly used to order models on an abstraction scale, not to systematically improve a model so that it becomes fit for its purpose. In terms of model quality, many criteria depend on the relevance of a construct or a statement. Still, evaluating objectively whether a given piece of information is relevant for an arbitrary purpose remains beyond the current state of the art.

Our research aims precisely at filling these gaps by developing methods to assess the relevance of a piece of information for a given purpose objectively. With this new ability, we can highlight irrelevant elements in a model and suggest missing ones, thereby assessing and improving

the model's level of abstraction. In the next section, we lay out our research goals and we define the research questions.

1.3 Research Goal and Questions

As motivated in the previous section, our research goal is to *find an objective measure of the adequacy of a model's level of abstraction for a given purpose and a technique to systematically improve this level of abstraction*. The quality attribute that we are interested in is an external attribute: it cannot be measured on the model alone. Indeed, it depends on both the model and the model's purpose. Assuming that the model has the right scope, we can define this attribute as follows:

Definition. A model is at the *right level of abstraction for a given purpose* if it is complete and confined with respect to this purpose. In other words, it contains all elements relevant for the purpose and it only contains these elements.

The London tube map example given by Kramer in [Kra07] suggests at least two reasons explaining why some models are at the wrong level of abstraction. One possible reason is that the creators of a model and its users do not share the same understanding of its purpose. Using the London tube map as a geographical map of London is a flagrant case of misunderstanding what the purpose of the map is.

This risk can be reduced by making the purpose of a model explicit. This explicit purpose has then two applications: For the modelers, it specifies the modeling task. For the model users, it documents what the model can be used for. Another reason is that a goal can be achieved in many possible ways and new techniques or ideas may improve the ways a goal is achieved. For example, Harry Beck drastically improved the way people can use the map for travelling with the tube by ignoring the geographical location of stations and by using a schematic representation for the tube network. As far as these issues are concerned, it is possible to detect and improve models at the wrong level of abstraction by confronting models to their usage.

In this context, the term confrontation does not suppose a duel, but rather a face-to-face comparison. Models are often built and used by different people, who may conceive its purpose differently. Ideally, we should compare these conceptions, identifying their similarities and their conflicts. However, this comparison is impossible, because these conceptions are held in the mind of different people. Instead, we confront the model with its usage to identify mismatches such as confinement or completeness issues. We choose the term confrontation over comparison to emphasize the fact that we bring the model and its usage together for the comparison and that we can improve the model by addressing the mismatches found in this comparison.

To attain our resarch goal, we perform an engineering cycle as presented in [WH06]. This cycle consists in three major steps: First, we

investigate the problem. Then, we devise and implement a solution for it. Finally, we evaluate the solution to investigate whether it indeed solves the problem. Thus, our first research question is explorative [ESSD08]:

Question 1. *How can we capture the purpose(s) of a model?*

Once we know how to characterize the purpose of a model, we need methods to assess and improve both the completeness and the confinement of a model with respect to this purpose. Therefore, the next research questions are design questions [ESSD08]:

Question 2. *How can we measure and improve the confinement of a model with respect to a given purpose?*

Question 3. *How can we measure and improve the completeness of a model with respect to a given purpose?*

Finally, we need to evaluate our work to know whether we attained our research goal. Hence, the last research question is an evaluation question [ESSD08]:

Question 4. *What is the impact of our approach on model quality?*

To evaluate our work, we follow a mixed design strategy [Rob11]: It consists of a flexible part and an inflexible part. In a flexible design strategy, the research design evolves during data collection. We use such a design strategy in the evaluations of each individual answers

to Questions 1, 2 and 3. For these evaluations, we use case studies and interviews as research methods. In contrast, we use a fixed design strategy to answer Question 4 where our approach is evaluated in its entirety. In this part, the research design is specified before the collection of data begins. We design a controlled experiment and present its results according to the guidelines provided by [WRH+00].

In the next section, we present our approach, which addresses the first three research questions.

1.4 MiRiA in a Nutshell

In this section, we present MiRiA, an approach that helps modelers in modeling at the right level of abstraction. Our model of abstraction is inspired by the work of Lindland *et al.* [LSS94]. We consider conceptual models as set of statements (see Figure 1.1). U is the (possibly infinite) set of all correct, but not necessarily relevant statements that can be made about an entity. Modeling consists of selecting some statements from the set U and making them explicit in m. The subset D of U is the (ideal) set of statements relevant for the purpose of the model being created.

In ideal cases, m and D are the same sets. When they are not, we can assess the deviation of m from D by using two well-known metrics from the Information Retrieval field: precision and recall. Precision

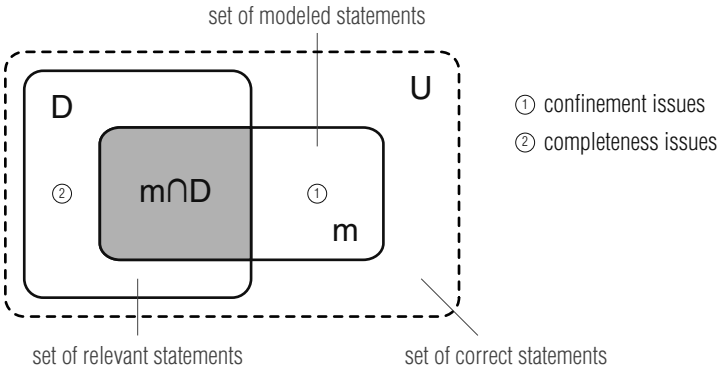


Figure 1.1: A model of abstraction.

measures the *confinement* of a model, the extent to which it contains relevant statements. In contrast, recall measures the *completeness* of a model, that is, the proportion of relevant statements that have actually been modeled.

$$\text{confinement} = \frac{|m \cap D|}{|m|}$$
$$\text{completeness} = \frac{|m \cap D|}{|D|}$$

These measures capture the common idea that a model at the right level of abstraction has all necessary details (it is complete) and does not contain any irrelevant information (the model is confined). Interestingly, a model can have at the same time completeness and

confinement issues, that is, it can have irrelevant details while missing relevant information (as it is the case for the model m in Figure 1.1). However, one limitations of these measures is that they do not distinguish whether superfluous or missing elements are abstraction issues or scoping issues. In this thesis, we assume that the models have the right scope and we focus on abstraction issues.

Defining the set D remains nevertheless challenging, as it is a semantic issue. Thus, we need a characterization of the purpose to reason about it and assess, objectively, whether a given statement is relevant for the purpose at hand. In this thesis, we propose to characterize the purpose of a model with the set of model operations to be performed on this model. We recall that by model operations, we mean any kind of inferences supported by a model such as simulations, queries, analyses and transformations.

Thesis Statement. *The purpose of a model can be characterized by the set of model operations that this model enables.*

This idea is similar to the idea of abstract data types [LZ74], where a data structure is defined by the set of operations that may be performed on it. However, abstract data types further specify some constraints on the effect of these operations. Besides, our statement is compatible with the model of modeling of Muller *et al.* [MFBC12] where intentions are formalized as sets. Furthermore, Rothenberg made a similar statement when he described the purposes of models in [Rot89]:

> The purpose of a model may include comprehension or manipulation of [the entity], communication, planning, prediction, gaining experience, appreciation, etc. *In some cases this purpose can be characterized by the kinds of questions that may be asked of the model.* For example, prediction corresponds to asking questions of the form "What-if...?" (where the user asks what would happen if the [entity] began in some initial state and behaved as described by the model). [Emphasis added]

To capture the purpose of a model, we developed the following method: First, the modeling purpose is captured using a goal-oriented technique. Then, these modeling goals are made operational by deriving a set of model operations and a metamodel supporting these operations. In software engineering, many techniques have been developed to elicit, document and operationalize goals. In MiRiA, we adapted and extended two of these methods to capture and operationalize purposes of models: the Goal-Question-Metrics paradigm (GQM) [Bas92] and the KAOS modeling language [vL09]. For both methods, the purpose of a model is eventually expressed as the set of model operations it enables.

Once a model's purpose has been captured and operationalized, it becomes possible to validate the model with respect to this purpose. In MiRiA, we measure model completeness and confinement with a novel concept called footprinting.

The *footprint* of a model operation is the set of model elements that it touches during its execution. Thus, the footprint contains all elements that affect the outcome of the operation. The footprint of a set of operations is the union of the footprints of each operation of the set. When the set of operations characterizes the purpose of the model m, the footprint is the intersection $m \cap D$: it is the set of elements that are both present in the model and used to fulfill its purpose.

MiRiA proposes two techniques to calculate the footprint of a model operation: dynamic and static footprinting. Dynamic footprinting computes the actual footprint of an operation by analyzing its execution trace. Thus, dynamic footprinting requires executing the operation. In contrast, static footprinting estimates the footprint of an operation by analyzing the source code of the operation. In a nutshell, it works as follows: The definition of the operation is first analyzed statically to extract its *metamodel footprint*, the set of all modeling constructs involved in that operation definition. The static (model) footprint is then obtained by filtering the input model, keeping only those elements that are instances of the metamodel footprint. This estimate is conservative, *i.e.*, the dynamic footprint is always a subset of the static one. In some experiments, we found that static footprinting is an efficient, yet precise estimate of dynamic footprinting.

The MiRiA approach works by confronting models to their usage. In this confrontation, footprints are compared with expected footprints

Table 1.1: Comparisons of footprints and the problems they may reveal.

With respect to a given set of model operations, compare the…	with the…	to detect…
footprint	expected footprint	confinement issues in the model
metamodel footprint	expected metamodel footprint	completeness issues in the model
footprint	expected footprint	problems in the model operations
metamodel footprint	specified metamodel footprint	problems in the model operations

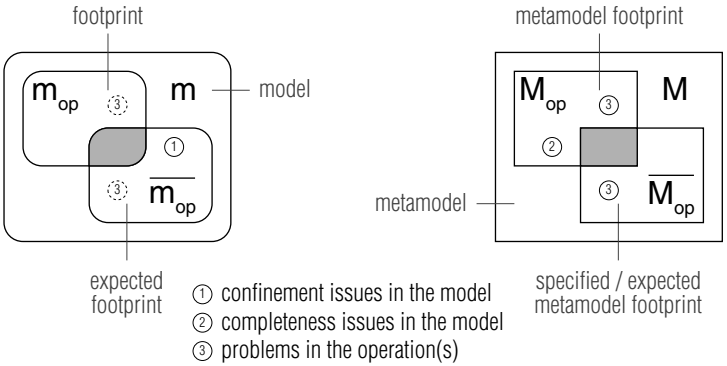


Figure 1.2: Validating models and operations with footprinting.

and metamodel footprints are compared with expected metamodels footprints. As illustrated in Figure 1.2, each of these comparisons may reveal problems in the model or in the operations. Table 1.1 summarizes these comparisons and the problems they may reveal.

Confinement is measured by analyzing elements that are not covered by the footprint. More precisely, the confinement of a model m is measured by comparing the footprint of a set of operations m_{op} to its expected footprint $\overline{m_{op}}$ (see left part of Figure 1.2). By default,

the expected footprint $\overline{m_{op}}$ is the whole model m. However, m may contain some elements that are meant for other operations or other purposes (*e.g.*, comments), and these elements should not be considered when assessing the confinement of a model with respect to a given set of operations. Elements that are not part of the footprint m_{op} but were expected to be $\overline{m_{op}}$ (① in Figure 1.2) are problematic: they were modeled for a given purpose, but they are not relevant for this purpose. Thus, they lower the confinement of the model with respect to this purpose.

Completeness can only be measured at the metamodel level (right part of Figure 1.2). For this, we compare the metamodel footprint M_{op}, which is computed during static footprinting, and the expected metamodel footprint $\overline{M_{op}}$, which is the part of the metamodel used in the model m. Constructs in the metamodel footprint M_{op} that are not covered by the expected metamodel footprint $\overline{M_{op}}$ (② in Figure 1.2) suggest that there are missing elements in m: The set of operations would use elements that are instances of these uncovered constructs if they were present in the model. Thus, a modeler can improve the model's completeness by adding such elements in it.

While we developed footprinting for measuring confinement and completeness, it has another important use in MDE: it can be used to validate model operations. Indeed, by comparing the metamodel footprint M_{op} of an operation with the specified or expected metamodel footprint $\overline{M_{op}}$, an engineer may find errors in the implementation

of the operation or may identify possible improvements to it (③ in Figure 1.2). In a case study, we found that MDE experts were able to detect bugs in model operations with the help of footprinting, but without reviewing the source code of the operations. Some of these errors can also be detected at the model level, provided that some test models are available.

The footprinting approach assumes that operations are formalized, so that they can be executed (dynamic footprinting) or analyzed (static footprinting). However, not all operations are formalized, especially those that are carried out mentally. For example, tourists visiting London do not follow an algorithm to find the shortest route between two stations on the London tube map. Nevertheless, if it is possible to define — manually — a metamodel footprint that supports these informal operations, then static footprinting can be applied to assess the completeness and the confinement of a model.

1.5 Contributions

In summary, the main contribution of this dissertation is the MiRiA approach to measure and improve the level of abstraction of a model with respect to a given purpose. The main components of MiRiA are:

1. Two approaches — Goal-Operation-Metamodel and Intentional Metamodeling — for capturing the purposes of models and operationalizing them with a set of model operations.

2. A method based on footprinting for improving the confinement and completeness of models with respect to a set of model operations.
3. A method for validating model operations using metamodel footprints.
4. Two techniques — dynamic and static footprinting — to calculate the footprint of a model operation.

1.6 Roadmap

The remainder of this dissertation consists of four chapters, each one being a scientific publication on its own, followed by a concluding chapter. Figure 1.3 displays the roadmap of the thesis, describing which concepts and topics are covered by which chapters.

Chapter 2 adapts two existing goal-oriented techniques to capture the purposes of models and operationalize them with a set of model operations (the GQM paradigm [Bas92] and the KAOS modeling language [vL09]). To demonstrate the feasibility of these approaches, we develop two examples: the London tube map introduced by Kramer in [Kra07] and the conversion of UML models to queueing networks proposed by Cortellessa *et al.* in [CM00]. This chapter addresses our first research question (RQ1) on capturing the purposes of models.

Chapter 3 introduces a method to analyze the confinement of a model with respect to a set of model operations. Thus, it addresses RQ2 on model confinement. We also present two techniques to compute the footprint of model operations: dynamic and static footprinting. Dynamic footprinting reveals the actual footprint after the execution of the operations, while static footprinting estimates the footprint without executing the operations. Based on an experiment involving 75 models and five model operations, we conclude that static footprints are precise, yet cheap to compute, estimates of dynamic footprints.

Chapter 4 extends Chapter 3 with two methods: a method to improve model completeness and a method to validate model operations. The former answers RQ3 on model completeness, while the latter demonstrate another application of footprinting. In some experiments, four MDE experts could detect some bugs in model operations without looking at their source code. Furthermore, we redefine static footprinting as a sequence of three model slicers expressed in Kompren, a DSL for defining model slicers [BCBB12].

Chapter 5 reports on a pair of controlled experiments involving students in an attempt to answer RQ4 on the benefits of our work on model quality. The results of these studies are inconclusive: A faulty experiment design may explain why they failed to demonstrate any significant benefits of static footprinting on model quality.

Finally, Chapter 6 concludes this thesis by summarizing our results and suggesting possible directions for future work.

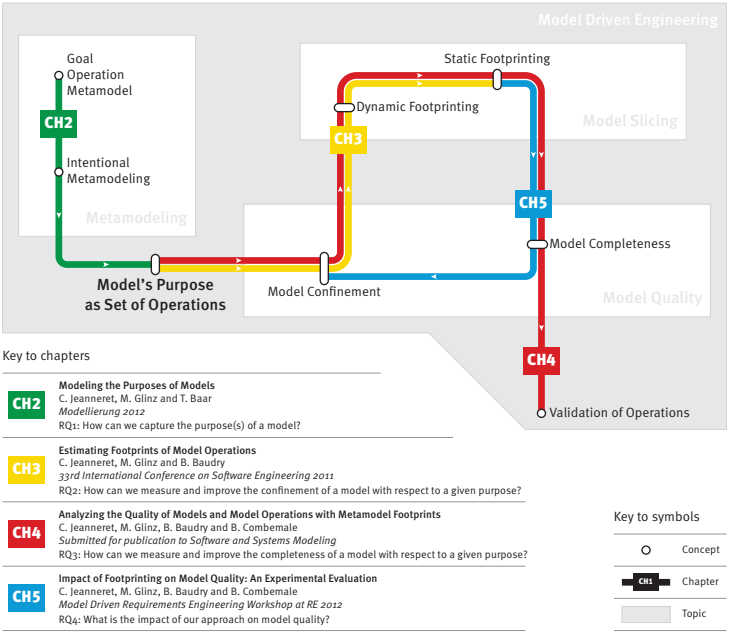


Figure 1.3: Thesis Roadmap.

Chapter 2

Modeling the Purposes of Models

Original publication:
Modeling the Purposes of Models
C. Jeanneret, M. Glinz and T. Baar
Modellierung 2012

Abstract

Today, the purpose of a model is often kept implicit. The lack of explicit statements about a model's purpose hinders both its creation and its (re)use. In this paper, we adapt two goal modeling techniques, the Goal-Question-Metric paradigm and KAOS, an intentional modeling language, so that the purpose of a model can be explicitly stated and operationalized. Using some examples, we present how these approaches can document a model's purpose so that this model can be validated, improved and used correctly.

2.1 Introduction

With the advent of Model Driven Engineering (MDE), models play a more and more important role in software engineering. Conceptually, a model is an abstract representation of an original (like a system or a problem domain) for a given purpose. One cannot build or use a model without knowing its purpose. Yet, today, the purpose of a model is often kept implicit. Thus, anybody can be mislead by a model if it is used for a task it was not intended for. Furthermore, a modeler must rely on his experience and his feelings to decide how much and which detail is worth being modeled. This may result in models at the wrong level of abstraction for its (unstated) purpose. Stating the purpose of a model explicitly is only a first step to address these issues.

Eventually, the purpose of a model can be characterized by a set of operations. There are two kinds of operations: (i) operations performed by humans to interpret (understand, analyze or use) the model and (ii) operations executed by computers to transform the model into another model (model transformations). Being able to express the purpose of a model with a set of model operations allows measuring how well a model fits its purpose. In previous work [JGB11a], we have made a contribution towards measuring the confinement of a model (the extent to which it contains relevant information) given the set of formal operations to be executed on it.

Having a set of operations is not enough, though: we must ensure that these operations can be performed on the model — no matter whether these operations are performed by humans or executed by computers. For this, we have to make explicit which information the operations need from the model and we have to determine which structures a model has to conform to. In other words, we need to state which elements the metamodel must contain for enabling the operations.

Our previous work assumes that these operations and these metamodels exist. This assumption may hold in an MDE context, but not in a wider context: Often, the purpose of a model is not even stated explicitly. Thus, there is a need for (a) methods to elicit and document modeling purposes in the first place and (b) methods to operationalize these modeling purposes systematically. In goal modeling, there are many approaches for these two tasks. However, these approaches were designed for other contexts than modeling.

In this paper, we adapt two of these goal modeling approaches for systematically deriving a set of model operations and associated metamodel elements from a qualitatively stated model purpose. First, we present Goal-Operation-Metamodel (GOM), a generalization of the Goal-Question-Metric (GQM) paradigm [Bas92]. Second, we propose to use KAOS [vL09] (a goal modeling language) as a metalanguage to create intentional metamodels.

The remainder of this paper is organized as follows. In the next section, we present the problem context of our work in more details.

In Section 2.3, we describe the Goal-Operation-Metamodel method and we present intentional metamodeling with KAOS in Section 2.4. We discuss our findings in Section 2.5 while Section 2.6 discusses related work. Finally, we conclude the paper in Section 2.7.

2.2 Problem Context

Many modeling theories distinguish two roles in the model building process: the *modeler* and the *expert*. Modeling is a collaborative activity involving a dialog among these two roles: The modeler elicits information about the original from the expert before formalizing it, while the expert validates the content of the model as explained by the modeler. These roles and the relationships are represented in Figure 2.1. Hoppenbrouwers *et al.* even consider the model as the minutes of this dialog [HPvdW05].

While building a model may be valuable on its own, the value of modeling consists of using the model as a substitute of the original to infer some new information about it. These inferences are made by the *interpreter* – a third role related to modeling. To achieve this, the interpreter performs various *model operations* on the model, like executing queries on it, extracting views from it or transforming it to other models or artefacts.

When describing the nature of modeling, Rothenberg listed the following purposes of models [Rot89]:

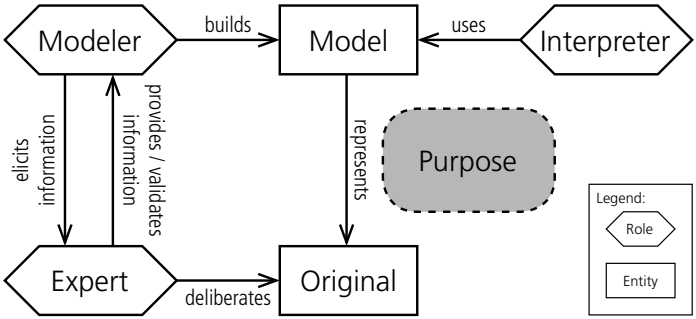


Figure 2.1: The roles involved in a modeling activity.

> The purpose of a model may include comprehension or manipulation of its referent [the original], communication, planning, prediction, gaining experience, appreciation, etc. In some cases this purpose can be characterized by the kinds of questions that may be asked of the model. For example, prediction corresponds to asking questions of the form "What-if...?" (where the user asks what would happen if the referent began in some initial state and behaved as described by the model).

A clearly stated modeling purpose can be used as a contract between the modeler and the interpreter. Establishing contracts is costly, as they must be negotiated and edited. Nevertheless, such a contract can be useful in two ways: First, as a specification for a model's purpose, it provides a strong basis on which the model can be validated. It

can also provide the modeler with some guidance for improving the model so that it reaches the right level of abstraction. Second, as a description of a model's purpose, it tells an interpreter whether the model at hand is fit for the intended use or, if several models are available, it helps him to choose which model will best fit his purpose.

In the vein of [LSS94], we consider a model as a set of statements M. For each modeling purpose, there is a set of relevant statements D. In an ideal case, the set D should correspond to the set of statements in the model M. When the sets M and D differ, we can quantify the deviation of M from D by using measures from the Information Retrieval field: precision and recall. Precision measures the *confinement* of a model, the extent to which it contains relevant statements. Recall, on the other hand, measures the *completeness* of a model, that is, the proportion of relevant statements that has actually been modeled. By measuring the confinement and completeness of a model, a modeler can assess how adequate is its level of abstraction for its purpose. Indeed, a model at the right level of abstraction for its purpose is both confined and complete ($M = D$).

However, defining the set D is challenging. In our previous work [JGB11a], we made a contribution towards measuring the confinement of a model given a set of operations that characterizes its purpose. When these operations are executed on a model, they navigate through its content and gather some information by reading some of its elements. The set of elements touched by a model operation

during its execution forms the *footprint* of that operation. Thus, the footprint contains all elements that affect the outcome of the operation. For a set of operations, the *global footprint* of the set of operations is the union of the footprints of each operation. This global footprint is the intersection $M \cap D$: it is the set of statements that are both present in the model and used to fulfill its purpose.

In this paper, we propose two approaches to operationalize a qualitatively stated modeling purpose into a set of model operations and their supporting metamodels. Instead of inventing new methods from scratch, we adapt two existing goal modeling techniques, GQM [Bas92] and KAOS [vL09], so that they can be used in a modeling context in addition to measurement and requirements engineering, respectively. To illustrate the use of these methods in modeling, we first present two examples.

2.2.1 Motivating Examples

In this section, we introduce two examples to motivate and illustrate our approaches to capture the purpose of a model. The first example is the *London Underground map*, used by Jeff Kramer in [Kra07] to highlight that the value of an abstraction depends on its purpose. The second example is related to Software Engineering, where an architect models her (or his) system according to the "4+1" viewpoints of Kruchten [Kru95] for making some performance analysis as described in [CM00]. We have used this example in our previous

paper to explain the various usage scenarios of model footprinting [JGB11a].

The London Underground Map

As most major cities, London has an underground railway system. To help its users to navigate in London with it, its operator, the *Transport for London* (TfL) company provides a map of this transit system. Figure 2.2 shows the evolution of the map along the years. In 1919 (Figure 2.2a), the map was a geographical map of London with the underground lines overlaid in color. In 1928 (Figure 2.2b), ground features like streets were removed from the map and the outlying lines were distorted to free some space for the congested center, making it more readable. The first schematic representation of the network appeared in 1933 (Figure 2.2c): the precise geographic location of stations is discarded; only the topology of the network is represented. The current map (Figure 2.2d) contains additional information such as the accessibility of stations, the connections to other transportations systems and fare zones.

In this example, the modeler is the employee of TfL designing the map. The expert is an employee of TfL who knows the underground network well. The interpreter is a user of the map. The map is used to plan trips in London, that is, the map must help travelers to answer the following questions: how to get from A to B? How much does it cost? How long does it take? Is that route accessible for disabled

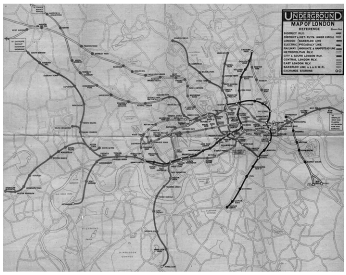
people? Interestingly, in this example, the people who use the model to plan their trip also use the modeled system to actually travel in London.

Performance Analysis on Software Architecture

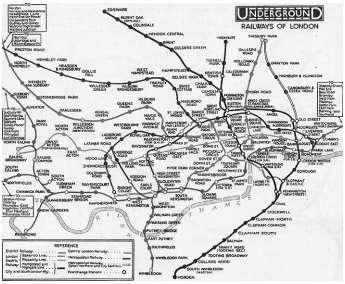
For the second example, we consider an architect analyzing the performance of a piece of software. To this end, she describes its architecture using the "4+1" view model proposed by Kruchten in [Kru95]: This view model includes (1) use case and sequence diagrams for the scenario view, (2) class diagrams for the logical view, (3) component diagrams for the development view, (4) activity diagrams for the process view and (5) deployment diagrams for the physical view. Her model is first transformed into an extended queueing network model (EQNM) as explained by Cortellessa et al. in [CM00]. Performance indicators are then measured on the EQNM. The architect wants the following questions to be answered: What is the response time and throughput of her system? Where is its bottleneck?

In this example, EQNMs can be seen as the semantic domain for architecture models written in UML. There are therefore two chained interpretations: the first interpretation translates a UML model into an EQNM, while the second interpretation analyses the EQNM. In this example, we focus on the translation from UML to EQNM.

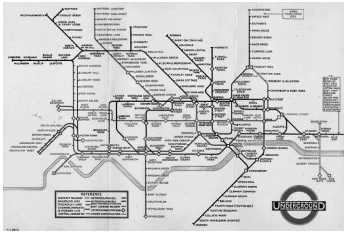
The architect plays all three roles in this example. As the architect of her software, she is the expert. As she creates the model, she is the



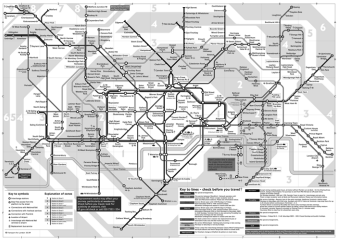
(a) Map in 1919



(b) Map in 1928



(c) Map in 1933



(d) Map in 2009

Figure 2.2: Maps of the London Underground.
(a), (b) and (c): ⓒTfL from the London Transport Museum collection
(d): ⓒTransport for London

modeler. As she uses the model for evaluating the performance of her software, she is the interpreter. However, there are two additional stakeholders involved in this example: Cortellessa and his team developed the analysis used by the architect, while Kruchten, by defining the "4+1" viewpoint, proposed a "contract" between the modeler and the interpreter.

Contrary to the previous example, the performance analysis is mostly automated. As the architect is only interested in its results, she may know little about the internals of the technique. Thus, the documentation of the analysis must state explicitly which information the analysis requires in input models.

2.3 Goal-Operation-Metamodel

GQM is a mechanism for defining and evaluating a set of operational goals using measurement [Bas92]. In the GQM paradigm, a measurement is defined on three levels:

- At the *conceptual* level, the goal of the measurement is specified in a structured manner: It specifies the purpose of the measurement, the object under study, the focus of the measurement and the viewpoint from which the measurements are taken.
- At the *operational* level, the goal is refined to a set of questions.
- At the *quantitative* level, a set of metrics is associated to each question so that it can be answered in a quantitative manner.

Our approach consists of using GQM for models other than metrics. According to Ludewig [Lud03], metrics are some kind of models. However, GQM has to be extended on its three levels to describe modeling purposes other than quantitative analysis. At the conceptual level, the goal template must support purposes like code generation[1] or documentation. At the operational level, the set of questions will be replaced by a set of (general) operations: Beside queries, one may need simulations and transformations to refine the goal stated at the conceptual level. Finally, the quantitative level becomes the *definable* level: metamodels replace metrics to support the model operations from the operational level. These operations will be run on conforming models in a similar way that questions can be answered with the value of a metric. Thus, we call this approach Goal-Operation-Metamodel (GOM).

2.3.1 GOM and the London Underground Map

Based on the GQM template described in [Bas92], we define the goal of the map as the following:

> Analyze *the London Underground*
> For the purpose of *characterization*

[1] Code generation, as an operation, is not supported when a model is at a conceptual level. Here, we consider code generation as the model's purpose to be described with GOM.

> With respect to *reachability* and *connectivity* of its stations
> From the view of *a traveler*
> In the following context: *the traveler may be a disabled person*, *the tube is part of a larger transportation system*, *the map is displayed on a screen or on paper in stations*

From this goal, we derive the following questions to be answered from the model:

(a) What is the shortest path between two stations?
(b) How much does it cost to travel along a given path?
(c) How long does it take to travel along a given path?
(d) Is a given path accessible to a disabled person?
(e) When traveling along a path, at which station to leave a train?
(f) When traveling along a path, in which train (line and direction) to enter?

Table 2.1 lists which questions are supported by the 4 versions of the map displayed in Figure 2.2. All maps can be used to find the shortest path between two stations and where to step in and step off trains. However, only the 2009 version fully supports disabled people and allows for computing the cost of a trip. Since it preserves the geographic location of stations, the map of 1919 can be used to estimate the time needed for a trip (without taking transfers into account).

Table 2.1: Operations supported by the different versions of the map.

Map	Path (a)	Cost (b)	Time (c)	Accessibility (d)	Step Off (e)	Step In (f)
1919	√	–	√	–	√	√
1928	√	–	–	–	√	√
1933	√	–	–	–	√	√
2009	√	√	–	√	√	√

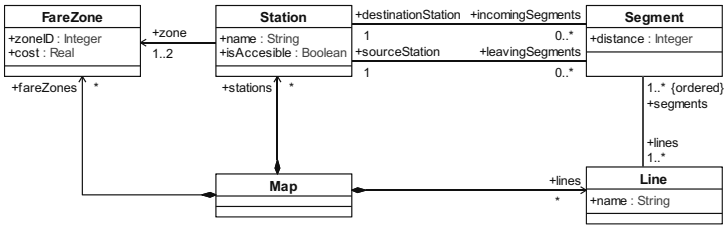


Figure 2.3: A metamodel for a map of the London Underground.

A map conforming to the metamodel depicted in Figure 2.3 could be used to answer all the questions characterizing the purpose of the map. Segments, lines and stations form the topology of the network, allowing a traveler for planning (question (a)) and executing (questions (e) and (f)) a trip with the Underground. Fare zones are involved in the computation of the cost of a trip (question (b)). The accessibility of a station serves for question (d) while the distance covered by a segment is needed to answer question (c).

In this example, questions are answered "mentally" by the travelers. Still, all these questions could be formalized with queries in OCL or operations in Kermeta [MFJ05]. For example, Listing 2.1 presents the operation computing the cost of a trip (encoded as a sequence of

segments) in Kermeta. This operation is defined for the metamodel presented in Figure 2.3.

```
// Compute the cost of a trip
operation cost(path: Sequence<Segment>): Real is do
  result := "0".toReal
  // First, collect all traversed zones
  var traversedZones: Set<FareZone> init Set<FareZone>.new
  path.each{seg |
    var src: Set<FareZone> init seg.sourceStation.zone
    var dst: Set<FareZone> init seg.destinationStation.zone
    var inter: Set<FareZone> init src.intersection(dst)
    if not inter.isEmpty
    then
      // Both stations are in the same zone
      traversedZones.addAll(inter)
    else
      // The segment traverses a boundary
      traversedZones.addAll(src)
      traversedZones.addAll(dst)
    end }
  // Second, sum the cost of each traversed zone
  traversedZones.each{z | result := result + z.cost}
end
```

Listing 2.1: Operation computing the cost of a trip.

2.3.2 GOM and the Performance Analysis on Software Architecture

In this example, we only consider the immediate goal of the model, which is the generation of an EQNM, and we leave the final goal (the performance analysis) out. Still, both goals could have been captured by GOM. Slightly adapting the GQM template [Bas92], the immediate goal of the model can be stated as follows:

> Analyze *the architecture of a software system*
> For the purpose of *generating an EQNM*
> With respect to *the scenario view* and *the physical view*
> as defined in [Kru95]
> From the view of *the software architect*
> In the following context: *the generation of an EQNM is explained in [CM00], this generation is automated*, *the generated EQNM will be used to analyze the performance of the architecture*

[CM00] describes, formally, the various steps in the generation of the EQNM from UML models:

(i) deduce a user profile from the use case diagram,
(ii) combine the sequence diagrams into a meta execution graph (meta-EG),
(iii) obtain the EQNM of the hardware platform from the deployment diagram and tailor the meta-EG into an EG-instance for that platform,

(iv) assign numerical parameters to the EG-instance, and
(v) assign environment based parameters to the EQNM, process the EG-instance to obtain software parameters before assigning them to the EQNM.

This chain of transformations requires information from the following UML diagrams: use case diagrams, sequence diagrams and deployment diagrams. The other diagrams of the "4+1" model — the class, the component and the activity diagrams — are not needed for this purpose.

While GOM allows stating the purpose of a model explicitly and operationalize it, goals expressed in GOM are not formal enough to be analyzed automatically, for example, to find conflicts among them. In the next section, we present how a model's purpose can be expressed in a goal-oriented modeling language.

2.4 Intentional Metamodeling

In the previous section, we presented a structured but informal way to specify a model's purpose. In this section, we introduce intentional metamodeling with KAOS, a goal modeling language designed for use in early phases of requirements engineering. A KAOS model consists of four interrelated views:

Goal modeling establishes the list of goals involved in the system. Refined goals and alternatives are represented in an AND/OR tree. Conflicts among goals are also represented in this diagram.

Responsibility modeling captures the agents to whom responsibility for (leaf) goal satisfaction is assigned.

Object modeling is used to represent the domain's concepts and the relationships among them.

Operation modeling prescribes the behaviors the agents must perform to satisfy the goals they are responsible for.

A goal can be refined into conjoined sub-goals (the goal is satisfied when all its sub-goals are satisfied) or into alternatives (the goal is satisfied when at least one of its alternatives is satisfied). Therefore, goals are represented as AND/OR trees in KAOS. In such a tree, the goals below a given goal explain how and how else the goal can be realized. On the opposite, goals higher in the hierarchy provide the rationale for a given goal, explaining why the goal is present in the model.

[vL09] provides a taxonomy of goals based on their types and their categories. There are two main types of goals: *behavioral* goals (such as Achieve, Cease, Maintain and Avoid goals) prescribe the behavior of a system, while *soft-goals* (such as Improve, Increase, Reduce, Maximize and Minimize goals) prescribe preferences among alternative systems. Similarly, there are two main categories of goals: *functional* goals (like Satisfaction [of user requests] or Information [about a system state] goals) state the intent behind a system service and *non-functional*

goals (like Usability or Accuracy) state a quality or constraint on its provision or its development. This taxonomy can be helpful for eliciting and specifying goals.

Goals are refined until they are assignable to a single agent. Leaf goals are then made operational by mapping them to operations ensuring them. Operations are binary relationships over systems states. They can be derived from the formal specification of goals or built from elicited scenarios. Finally, a conceptual model gathers all concepts (including their attributes and the relationships among them) involved in the definition of goals and operations.

We use KAOS as a metametamodel and not as a metamodel as it was initially designed for: In our approach, KAOS models are metamodels. Goals depict the modeling purposes. Operations prescribe the operations that can be executed on models and the conceptual model defines the abstract syntax of the language. Thus, a metamodel written in KAOS specifies many aspects of a modeling task: it states the purpose and intended usage of models as well as their structure. In the remainder of this section, we present KAOS metamodels for our examples.

2.4.1 Intentional Metamodeling and the London Underground Map

A KAOS metamodel of the London Underground map is presented in Figure 2.4. The main goal of the map is to provide travelers with a

means to understand how to travel from a station A to another station B successfully. To achieve this, the map must satisfy the following sub-goals: to help travelers to plan their trip, to help them to buy the right ticket for it and to help them for the navigation, that is, to prevent them from getting lost during their travel.

These goals are operationalized through the following operations performed by the traveler: find the shortest path between stations A and B, compute its cost (by summing the fares of traversed fare zones) and carry out the plan by riding on the right line and connecting on the right station. As we did in Section 2.3.1, we can define these operations formally and derive a metamodel to support them. For space reasons, this metamodel is not included in Figure 2.4, but it presented in Figure 2.3.

2.4.2 Intentional Metamodeling and the Performance Analysis

We present an intentional metamodel of the performance analysis in Figure 2.5. The final goal of the architect is to analyze the performance of her architecture. This goal has been refined to three sub-goals: First, performance models are generated automatically from some UML diagrams. Then, these performance models are parameterized and solved. For space reasons, we did not further elaborate these two latter goals. We also considered UML diagrams as atoms, ignoring their internal elements such as actors, messages and nodes.

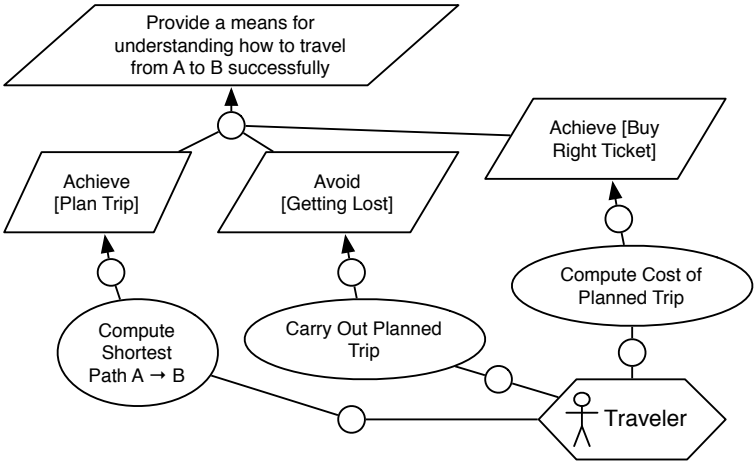


Figure 2.4: A KAOS metamodel for the London Underground map.

A computer is responsible for the generation of performance models. This goal is operationalized with four automated operations: generate the user profile from the use case diagram, generate the meta-EG from sequence diagrams, instantiate the meta-EG into an EG-instance with the help of the deployment diagram and generate an EQNM from the deployment diagram. These operations correspond to the first three steps described in [CM00]. The last two steps are captured in the two remaining goals, parameterize and solve the performance models.

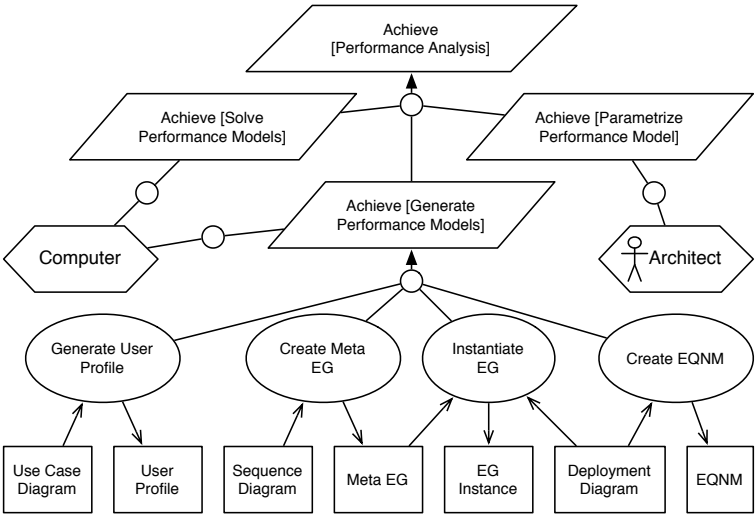


Figure 2.5: A KAOS metamodel for a performance analysis.

2.5 Discussion

This paper is an initial contribution towards the modeling of models' purposes. For this, we have adapted two existing goal modeling approaches and applied them to two modeling tasks, demonstrating the feasibility of such metamodeling.

In the remainder of this section, we compare the two approaches presented in this paper, GOM and intentional metamodeling. We also discuss the benefits and difficulties related to these approaches.

2.5.1 Comparison GOM and Intentional Metamodeling

Contrary to GOM, intentional metamodeling with KAOS can capture the complete rationale behind the creation and the use of a model. As explained in Section 2.4, goal models are organized in AND/OR trees. By navigating the tree from an element upwards, a modeler can find the rationale explaining a given operation, meta-class or modeling purpose. Likewise, but using downward navigation, the modeler can figure out how a model purpose is realized by looking at its sub-goals or its alternatives.

In this paper, we only presented semi-formal KAOS models. However, these models can be completely formalized and thus are amenable to automated analysis, including the verification of goal refinements or

the derivations of goal operationalizations [vL09]. The weaknesses of KAOS lie in the cost and difficulties of formalizing goals and operations. In comparison, GOM is a semi-formal approach. It only provides templates for stating modeling purposes and guidelines for deriving questions from this purpose. Future research should explore under which conditions a low or a high level of formality is preferred or required.

We have presented GOM and intentional metamodeling as two different approaches, because they come from different field: software measurement and early requirements engineering, respectively. Future work may integrate these two approaches, combining the ease of use of the templates and guidelines of GOM and the formality of KAOS.

2.5.2 Benefits and Limitations

Stating a model purpose and making it operational allows for measuring the fitness of a model for this purpose. A model is complete if it contains all the elements necessary to fulfill its goals. Conversely, a model element is pertinent if it contributes to the satisfaction of at least one goal. Confined models only contain pertinent elements. With a formal KAOS model, it is possible to measure these qualities objectively by establishing satisfaction arguments.

However, eliciting a model's purpose and elaborating it has a cost. The benefits must be higher than the costs if the practice is to be

adopted by practitioners. Models are like systems. Making explicit requirements about models (such as stating their purpose) aims at reducing the risk of creating the wrong models. Models at the wrong level of abstraction have consequences ranging from small annoyances for their interpreters to the impossibility of fulfilling the purposes they were made for.

Furthermore, goal modeling is difficult. First, many modelers are not experienced in intentional modeling. Courses on Software Engineering or Modeling typically cover data, behavior and process modeling languages but leaves out goal modeling. Thus, (intentional) metamodelers will be rare in the near future. Second, goal models grow rapidly as goals are refined and alternatives are identified.

2.6 Related Work

In this section, we present the state of the art in metamodeling and model quality and we discuss its limitations. For van Gigch [vG91], a metamodel should cover many aspects of modeling, not only "data" metamodeling (the syntax of the language). In this vein, Kermeta proposes to metamodel the behavior of models, so that the operational semantics of models can be specified [MFJ05]. In this paper, we go one step further by metamodeling modeling agents and their goals.

In their model of modeling [MFBC12], Muller et al. place the intention of a model at the heart of their notation. They define intention as follows:

> The intention of a thing thus represents the reason why someone would be using that thing, in which context, and what are the expectations vs. that thing. It should be seen as a mixture of requirements, behavior, properties, and constraints, either satisfied or maintained by the thing.

In their notation, intentions are considered as sets and thus represented as Venn diagrams. While this notation allows representing the intersection and the overlap among intentions, it does not allow representing the internal content of the intention behind a model. The focus of our paper is to represent this intention, so that its modelers and its interpreters can agree and reason on it.

For Nuseibeh et al. [NKF93], a *viewpoint* consists of (1) a style (the modeling language and its notation), (2) a work plan describing the development process of the viewpoint including possible consistency check or construction operations, (3) a domain defining the area of concern with respect to the overall system, (4) a specification describing the viewpoint's domain using the viewpoint's style (in other words, the view of the system from the viewpoint) and (5) a work record keeping track of development history within the viewpoint. According to the IEEE 1471 standard, a viewpoint captures the conventions

for constructing, interpreting and analyzing a particular kind of view. Thus, a viewpoint defines — among others — modeling languages, model operations that can be applied to views and stakeholders whose concerns are addressed in the views. Viewpoints define the various views (and their relationships) in a software specification or in an architecture description, thus, they provide the modeler with guidelines on what they are expected to model. However, we are not aware of guidelines to define these viewpoints, nor techniques to validate that a view actually satisfies the needs of the stakeholders using it.

In [MDN09], Mohaghegi surveyed frameworks, techniques and studies of model quality in model based software development. They identified 6 quality goals: correctness, completeness, consistency, comprehensibility, confinement and changeability. Manual reviews [LSS94] and metrics [BB06] can be used to assess and improve the confinement and completeness of models. However, these techniques are either bound to a given modeling language and a given process [BB06] or must be tailored for the modeling task at hand [LSS94]. In comparison, intentional metamodeling and GOM are not bound to any specific language or process. Because they document the purpose of models, goals expressed and operationalized in GOM or KAOS may serve as basis to derive checklists, guidelines and metrics for validating models.

In previous work [JGB11a], we propose and compare two methods to compute the *footprint* of an operation – the set of all information used by the operation during its execution. *Dynamic footprinting*

reveals the actual footprint of an operation by tracing its execution on the model. In contrast, *static footprinting* estimates footprints by first analyzing, statically, the definition of the operation to obtain its *static metamodel footprint*, the set of all modeling constructs (*i.e.*, types, attributes and references) involved in this definition. The model footprint can then be estimated by selecting only those model elements that are instances of elements in the metamodel footprint. In this previous work, we assumed that the purpose of a model can be characterized by the set of operations being carried on it and that these operations were formally defined. These assumptions are reasonable in a MDE setting. Still, in this paper, we are interested in methods to specify an arbitrary model purpose and, if possible, to refine this purpose into a set of operations whose footprints can be looked at. In other words, the focus of this paper is the elicitation, documentation and operationalization of modeling purposes. The operationalization produces metamodels and operations that can be used, accessorily, as input for model footprinting.

In addition to GQM and KAOS, there are other goal oriented techniques and methods for requirements engineering, such as i* [Yu97] and Tropos [CKM02]. While we could have selected these approaches to capture and analyze the purposes of models, we chose KAOS and GQM instead for their strong focus on the operationalization of goals.

2.7 Conclusion

One cannot build a model without knowing its purpose, and one must not use a model for purposes it is not fit for. Despite its importance, the purpose of a model is often kept implicit.

In this paper, we adapted two existing goal modeling approaches — GQM [Bas92] and KAOS [vL09] — to capture the purpose of a model and operationalize it into a set of operations and a metamodel. With these elements in hands, it is possible to measure how fit a model is for the purpose. We demonstrated the feasibility of the approaches by applying them to two examples.

These early results are promising, but the benefits of such intentional metamodels remain to be established empirically (*e.g.*, with industrial case studies). With the experience gained in modeling the purpose of models, we can elaborate the templates and adapt the guidelines offered by KAOS and GQM in more detail. In this vein, further research could define a profile for KAOS and develop specific analysis for intentional metamodels.

For the first time, goal modeling techniques were applied to modeling itself, raising many open issues: What is the source of modeling goals, that is, who are the metaexperts? Do intentional metamodels help in model management and model reuse?

Chapter 3

Estimating Footprints of Model Operations

Original publication:
Estimating Footprints of Model Operations
C. Jeanneret, M. Glinz and B. Baudry
International Conference on Software Engineering 2011

Abstract

*When performed on a model, a set of operations (*e.g.*, queries or model transformations) rarely uses all the information present in the model. Unintended underuse of a model can indicate various problems: the model may contain more detail than necessary or the operations may be immature or erroneous. Analyzing the footprints of the operations* — i.e.,

the part of a model actually used by an operation — is a simple technique to diagnose and analyze such problems. However, precisely calculating the footprint of an operation is expensive, because it requires analyzing the operation's execution trace.

In this paper, we present an automated technique to estimate the footprint of an operation without executing it. We evaluate our approach by applying it to 75 models and five operations. Our technique provides software engineers with an efficient, yet precise, evaluation of the usage of their models.

3.1 Introduction

A model is an abstract representation of an original for a given purpose. Software engineers use many kinds of models in various contexts, ranging from informal models sketched on a whiteboard to executable models deployed in a production environment. In this paper, we are interested in models used for analysis and generation purposes. Such models are mainly used as input to various *model operations* like queries, simulations, views extractions or model transformations. Increasingly, model operations are automated to improve both the productivity of engineers and the reliability of the analysis [Sel03]. To support this automation, models must be machine-processable, *i.e.*, models must be expressed in a modeling language with a formal syntax, such as UML, Adora [GBJ02] or Petri nets. These modeling languages are defined by a metamodel.

When performed on a model, an operation computes new information based on the information stored in the model. During its execution, it navigates through the content of the model and gathers some information by reading some of its elements. The set of elements touched by a model operation during its execution forms the *footprint* of that operation. Thus, the footprint contains all elements that affect the outcome of the operation, as long as this operation is deterministic and does not use data other than those contained in the input model.

Footprints rarely cover models completely. The ratio between the size of a footprint and the size of the model quantifies the *model usage* of an operation. This measure can be used as a diagnosis to detect problems with the model's scope or its level of detail or the presence of faults in an operation. Identifying the footprint of an operation in a model further helps engineers in solving these problems by highlighting elements in the model that were used by the operation and separating them from the elements that remained unused.

Actual footprints can be computed with a dynamic analysis [CH06], by tracing the execution of an operation on a model. Unfortunately, the dynamic nature of footprints makes them at least as expensive to compute as performing the operation. To be effective and practical, a diagnosis must be available when it is needed and should be inexpensive to perform. Therefore, obtaining the footprint of an operation without having to execute it would be valuable to software engineers.

In this paper, we motivate the use of footprints in modeling activities and then concentrate on a novel approach for effectively and efficiently *estimating footprints* without executing the operation. In a nutshell, our approach works as follows: By analyzing the formal definition of an operation, we establish its metamodel footprint, the set of modeling constructs involved in this definition. The model footprint can then be estimated by selecting only those model elements that are instances of elements in the metamodel footprint. We call such a footprint estimate the *static footprint*, whereas the actual footprint

is called the *dynamic footprint*. We have chosen this terminology in analogy to the static and dynamic analysis in the program analysis field [Ern03].

We have evaluated our approach with 75 real-world models which are actually metamodels written in Ecore. This is no threat to the validity of our results, because our evaluation concentrates on the quality of static footprints in comparison to dynamic ones where the kind of models used does not matter. First, we illustrate the meaningfulness of footprinting in a case study where we present 5 representative model operations that do not use all the information contained in these models, even when they are executed all together. Then, we evaluate empirically the quality of the estimation made by static footprinting. In experiments involving these 75 models and 5 (+ 1 combining these 5 operations) model operations, we demonstrate that static footprints (a) are indeed conservative estimates of dynamic footprints, (b) are very precise with respect to dynamic footprints and (c) are much faster to compute than dynamic footprints.

The remainder of the paper is organized as follows. In the next section, we present the various uses of footprinting during modeling activities. In Section 3.3, we introduce a measure for assessing the model usage of an operation. Section 3.4 presents static and dynamic footprinting, while Section 3.5 evaluates our approach with a case study and some experiments. We discuss our contribution in Section 3.6 before contrasting it with related work in Section 3.7.

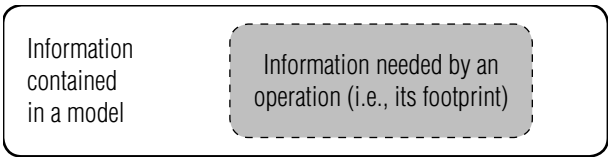


Figure 3.1: Gap between a model and its usage.

3.2 Motivation

A typical modeling assignment consists of an original to be modeled, a modeling purpose and a modeling language, *e.g.*, UML, which is defined by a metamodel. Frequently, the purpose of a model can be characterized in terms of an operation or a set of operations to be performed on that model, particularly in the context of model-driven development. In this case, a gap between the information contained in the model and the information required by the set of operations that will be executed on the model (*cf.* Figure 3.1) is an indicator of a problem.

For example, assume that a modeler needs to create a UML model with the purpose of deriving a performance model based on queueing networks as presented in [CM00]. Assume further that the modeler actually creates a model that fully documents the architecture of some software according to the "4+1" view model [Kru95]. Among other diagrams, this model includes (1) use case and sequence diagrams for the scenario view, (2) class diagrams for the logical view, (3) component diagrams for the development view, (4) activity diagrams for the

process view and (5) deployment diagrams for the physical view. Such a model would contain too much information for the operation presented above, because this operation only uses information from use cases diagrams, sequence diagrams and deployment diagrams.

Models with excessive information with respect to an operation require more time and resources for their creation and their processing by the operation than necessary. As illustrated in Figure 3.1, the modeler can identify information that is excessive with respect to an operation by establishing the footprint of this operation. An analysis of the size and extent of the actual footprint of an operation (or a set of operations) in comparison to the expected size and extent can reveal several problems: (i) the model may have the wrong scope, (ii) it may contain information that nobody needs or it may be on the wrong level of abstraction (*i.e.*, it contains unnecessary details), (iii) it serves more or other purposes than initially intended, (iv) the operation(s) concerned may be erroneous or immature.

In the example given above, the presence of nodes in the deployment diagram that are not involved in any scenario (interaction) is a problem of scoping (i). Furthermore, the class diagram of the logical view is completely ignored by the operation (ii). Problem (iii) can be illustrated with the presence of comments meant to improve the understandability of the model. Finally, the operation ignores elements from the activity diagrams. Still, these diagrams could provide useful information about workload derivation [CM00], suggesting a possible improvement to the operation (iv).

Beyond problem identification, calculating the footprint of an operation may also serve for generating a dynamic view of a model that contains only those parts of a model that are relevant for a given operation. This generated view can then serve for impact analysis: Any change made in this view may impact the results of an operation. Conversely, to improve the unsatisfactory output of an operation, the modeler can focus on the elements from this view.

Note that there is another form of gap between a model and its usage: A model may lack some important information for a given operation. In this case, the operation may fail or its result may be imprecise. This paper is concerned with the excessive part of a model with respect to a set of operations and leaves the missing part for future work.

For management purposes, it is sufficient to know the size of a footprint, so that it can be compared to the size of the model. By assessing the model usage of operations systematically, a project manager can locate gaps between models and their usage and undertake appropriate actions. Then, establishing the footprint can further help in reducing these gaps. In the next section, we propose a measure to quantify the usage of a model by one or more operations.

3.3 Measuring Model Usage

In order to quantify the usage of a model by a model operation, we first need a metric for the size of a model or model footprint. For our

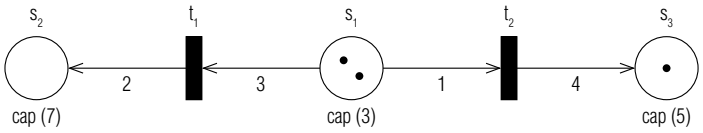


Figure 3.2: A Petri net.

purpose, it suffices to count the number of elements present in the model or footprint.

To illustrate the further discussion, we introduce a running example: A simple Petri net model is given in Figure 3.2. Figure 3.3 shows a metamodel defining Petri nets of the kind given in Figure 3.2. Basically, such a *metamodel* is a set of types and features. *Types* can either be (meta)classes or data types (enumerations or primitive types like integer). *Features* are divided into *structural features* (attributes within classes or references to other classes) and *behavioral features* (operations). For example, in the metamodel presented in Figure 3.3, `Transition` and `TransitionKind` are types, `capacity` and `source` are structural features and `fire()` is a behavioral feature.

In terms of its metamodel, a model can be regarded as a set of objects and settings. Every *object* is an instance of a class defined in the metamodel. A *setting* represents a value or a set of values held by an object for a structural feature. Settings can be set to a given value, reset to the feature's default value or unset. The set of settings in an object forms the *state* of this object. In our Petri net example (Figure 3.2),

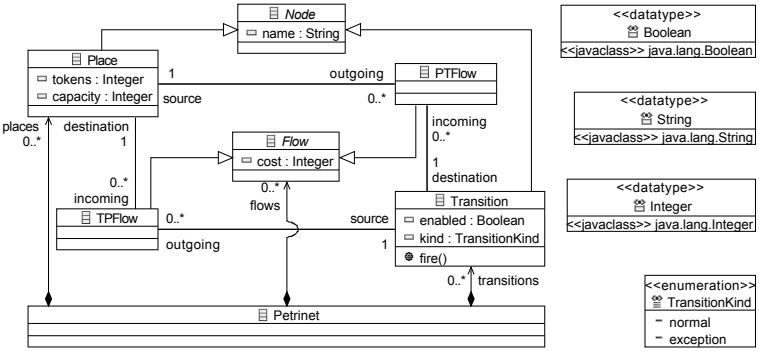


Figure 3.3: A metamodel for Petri nets.

place s_1 has the following five settings: (name = "s_1", tokens = 2, capacity = 3, incoming = nil, outgoing = $\{s_1t_1, s_1t_2\}$).

We define two size metrics by counting (i) the *number of objects* a model or footprint contains and (ii) the *total number of settings* present in the model or footprint. Thus, the size of our example Petri net is 10 objects and 40 settings: it contains 1 object of type `Petrinet` with 3 settings, 3 places (each object of type `Place` has 5 settings), 2 transitions (5 settings per transition) and 4 flows (3 settings per flow).

The usage of a model by a given model operation (or set of operations) can now be formally defined as follows. First, we recall that the footprint of an operation is the set of elements touched by this operation, *i.e.*, the footprint consists of those parts of a model that are actually needed by the operation. If we have a set of operations,

we define the footprint of this set as the union of the footprints of each individual operation. We can now define model usage with the *usage ratio* η, which has two components: η_o is based on the number of objects, while η_s is counting the number of settings.

$$\eta_o = \frac{\text{\# objects in footprint}}{\text{\# objects in model}}$$
$$\eta_s = \frac{\text{\# settings in footprint}}{\text{\# settings in model}}$$

The higher these values, the more a model operation uses the elements in the input model. In the extremes, $\eta = 1$ means that the operation uses the model completely, while $\eta = 0$ indicates an empty footprint. We explain how the footprint of an operation can be determined in the next section.

3.4 Calculating Footprints

The actual footprint of an operation in a model is the *dynamic footprint* (*cf.* the definition in the introduction). An algorithm for calculating a dynamic footprint is given in Section 3.4.1. The calculation is not difficult, but it is expensive, because it requires the execution of the operation on the model under analysis. Therefore, we introduce *static footprints* which can be calculated much faster and provide a

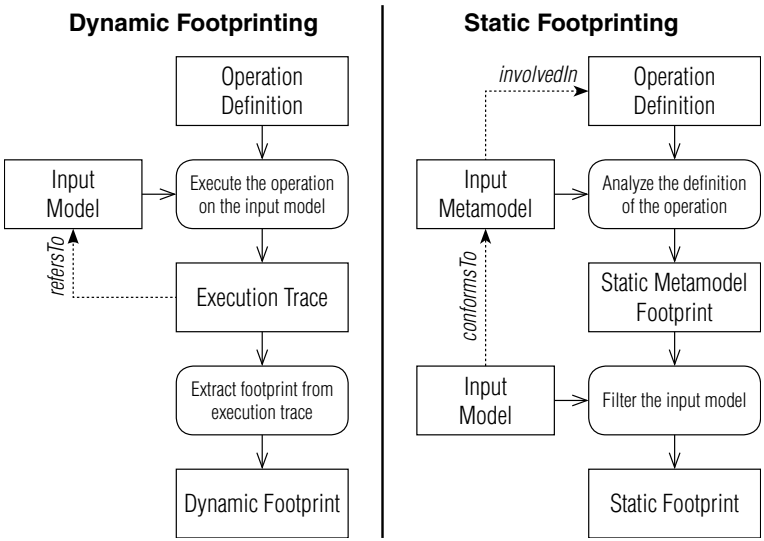


Figure 3.4: Dynamic and static footprinting.

conservative footprint estimate (*i.e.*, the dynamic footprint is always a subset of the static one). The calculation of static footprints is presented in Section 3.4.2. Figure 3.4 illustrates both approaches and Section 3.4.3 compares them.

3.4.1 Dynamic Footprinting

Dynamic footprinting (left part of Figure 3.4) is the most intuitive method to reveal the footprint left by an operation. It consists of executing the operation on the model while recording a trace of this execution. An *execution trace* is a set of events that occurred during

the execution of an operation. We are interested in two kinds of events: accesses to the states of objects and invocations of operations on objects. We can then establish the dynamic model footprint of an operation from its execution trace with a simple book-keeping algorithm.

This algorithm uses an array in two dimensions to keep track of relevant objects and settings. Each row represents an object, while each column represents a structural feature of the input metamodel. This array is initially empty. For every invocation of an operation *op* on an object *obj* in the execution trace, we create a row for *obj* (unless such a row already exists). Similarly, for every access to a structural feature f of an object *obj*, we create a row for *obj* (unless such a row already exists) and a column for f (unless such a column already exists) and we mark the cell at the intersection of the row and the column. Additionally, if f is a reference, we insert all referenced objects into the table. Once the trace has been fully analyzed, the dynamic footprint is the set of objects present in the array and the set of settings marked in the array.

To illustrate dynamic footprinting, we use our running example and calculate the footprint of a query that extracts the names of enabled transitions in Petri nets. This operation can be formally defined in OCL [OMG12] as follows:

```
context Petrinet::namesOfEnabledTransitions():
    Set(String)
  body:
```

```
    self.transitions->select(t: Transition |
        t.enabled)->collect(t: Transition | t.name)
```

The attribute `enabled` (see metaclass `Transition` in Figure 3.3) is a structural feature whose value is derived from other values. A transition is enabled if (a) there are enough tokens in source places and (b) destination places have enough capacity to store additional tokens. It can be formally defined as follows:

```
context Transition::enabled: Boolean
  derive:
    self.incoming->forAll(f: PTFlow | f.source.tokens
        >= f.cost) and self.outgoing->forAll(f: TPFlow
        | f.destination.capacity - f.destination.tokens
        >= f.cost)
```

Executing this operation on the Petri net of Figure 3.2 yields an execution trace consisting of 24 accesses to objects (assuming that OCL expressions are not evaluated lazily). An excerpt of this trace is shown on the left side of Figure 3.5, while the result of the trace analysis is presented on the right side. During its execution, the operation has touched 10 objects and 21 settings. The operation therefore uses all objects in the model ($\eta_o = 1$) but only half of its settings ($\eta_s = 0.52$).

3.4.2 Static Footprinting

Frequently, model operations concentrate on certain kinds of model elements and do not touch the rest of the model. In metamodeling

#	object	feature
1	p: Petrinet	*transitions*
2	t1: Transition	*enabled*
3	t1: Transition	*incoming*
4	s1t1: PTFlow	*source*
5	s1: Place	*tokens*
6	s1t1: PTFlow	*cost*
7	t1: Transition	*outgoing*
8	t1s2: TPFlow	*destination*
9	s2: Place	*capacity*
10	t1s2: TPFlow	*destination*
11	s2: Place	*tokens*
12	t1s2: TPFlow	*cost*
13	t2: Transition	*enabled*
[…]		
24	t2: Transition	*name*

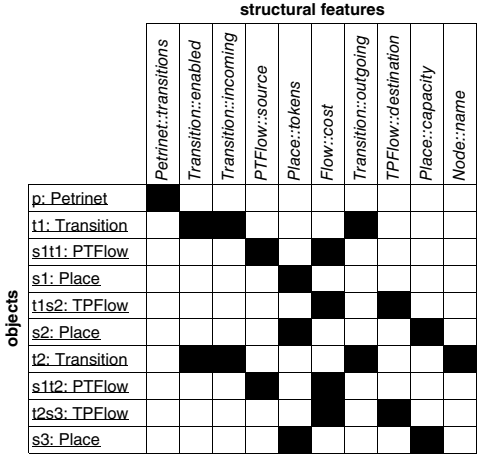


Figure 3.5: Dynamic footprinting of the Petri net example.

terms, this means that not all elements in the metamodel will be relevant for the operation. This is particularly the case with modeling languages that are designed to cover a large variety of modeling purposes such as UML. For example, [CM00] presents an operation which derives a performance model based on queueing networks from a UML model. While such a model may document many aspects of a software system, this operation only considers use case diagrams, sequence diagrams and deployment diagrams.

Static footprinting exploits this observation: it estimates the actual (dynamic) footprint by extracting those elements from the metamodel that are relevant for the operation (yielding a *metamodel footprint*) and then filtering the model by selecting only those model

elements that are instances of the metamodel footprint (right part of Figure 3.4).

The static metamodel footprint of an operation is extracted through a static analysis of its formal definition. This analysis consists of collecting metamodel elements involved in the definition of the operation. For operations written using declarative languages (such as triple graph grammar [Sch94]), this analysis is performed on the left-hand side part of every rule. If the operation is defined in an imperative language, metamodel elements are collected along the *control flow graph* of the operation, that is, the set of all its possible execution paths.

More precisely, for every expression involving features, we add the feature and the type of the object on which this feature is "applied" (accessed or invoked) to the static metamodel footprint (unless these metamodel elements come from the language's library and not the input metamodel). For invocations of behavioral features (operations defined in classes), we also add the types of their parameters and their return types. For accesses to structural features, we include the type of this feature in the static metamodel footprint. Furthermore, when a class is added to the static metamodel footprint, we insert all its subclasses as well to make subsequent model filtering simpler.

Table 3.1 illustrates the analysis of the query introduced in Section 3.4.1 which extracts names of enabled transitions in Petri nets. The left column lists all expressions found in its control flow graph

Table 3.1: Static metamodel footprint of the Petri net example.

Expression	Feature	Types
self.transitions	*Petrinet::transitions*	Petrinet, Transition
t.enabled	*Transition::enabled*	Transition, Boolean
self.incoming	*Transition::incoming*	Transition, PTFlow
f.source	*PTFlow::source*	PTFlow, Place
f.source.tokens	*Place::tokens*	Place, Integer
f.cost	*Flow::cost*	PTFlow, Integer
self.outgoing	*Transition::outgoing*	Transition, TPFlow
f.destination	*TPFlow::destination*	TPFlow, Place
f.destination.capacity	*Place::capacity*	Place, Integer
f.destination	*TPFlow::destination*	TPFlow, Place
f.destination.tokens	*Place::tokens*	Place, Integer
t.name	*Node::name*	Transition, String

that involve a feature from the Petri net metamodel. The second column lists the features involved in these expressions while the right column lists the types involved in the corresponding expression. Thus, the static metamodel footprint in our example is the set of features and types in Table 3.1.

To visualize this static metamodel footprint, we can create a view of the metamodel specifically tailored for the query by pruning the complete metamodel [SMBJ09]. Figure 3.6 presents such a view of the Petri net metamodel (the complete metamodel is illustrated in Figure 3.3). This view contains all metamodel elements from Table 3.1, and, additionally, the classes `Node` and `Flow`. They have been included in this view, because some of their features — `name` and `cost` — are part of the static metamodel footprint.

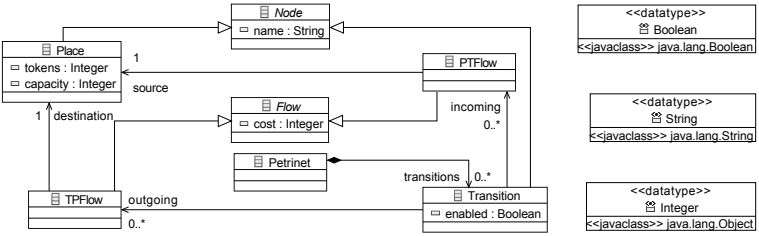


Figure 3.6: Static footprint metamodel for the Petri net example.

Once the static metamodel footprint has been computed, it can be used to create static (model) footprints by filtering input models. This filtering is straightforward: we first remove objects whose (meta)class is not part of the static metamodel footprint and, for every remaining object, we unset all its settings whose feature is not part of the static metamodel footprint. In our example, the static (model) footprint contains 10 objects and 26 settings: 1 petrinet (1 setting), 3 places (3 settings each), 2 transitions (4 settings each) and 4 flows (2 settings each). For comparison, the complete model has 10 objects and 40 settings. Thus, the model usage of the Petri net example is estimated to $\widehat{\eta_o} = 1$ and $\widehat{\eta_s} = 0.65$.

3.4.3 Comparison

In this section, we compare dynamic and static footprinting and discuss their limitations. The major difference between them is that dynamic footprints can be obtained only *after* the execution of the

operation, while static footprint can be computed *without* executing the operation. This difference has major impacts on both the effort required by these approaches and their precision.

Static footprinting requires less effort than dynamic footprinting. The static metamodel footprint must only be computed once for a given operation definition. It can then be used to filter the static footprint for any model. On the opposite, dynamic footprinting requires executing the operation on every model. Thus, it is preferable to use static footprinting if footprints are to be computed regularly (for example, when monitoring the model usage along the evolution of a model).

On the other hand, dynamic footprinting reveals the actual footprint of an operation with respect to an input model, while static footprinting estimates footprints based on two approximations. First, it considers all possible execution paths of an operation, not only the actual execution path. For operations implemented as graph transformations, it takes all rules into account, including those that are not actually used. Second, it essentially works with types, not with individual instances. This means that static footprinting overlooks all conditions defined in terms of object states. In our example, the names of all transitions are part of the static footprint, while only the names of enabled transitions are included in the dynamic footprint. Even worse, `name` being an attribute of `Node`, names of places are also part of the static footprint. Therefore, it may happen that static footprinting is not precise enough for assessment purposes. In the

next section, we evaluate the precision of static footprinting and compare the efforts required by both footprinting approaches on a sample of models.

3.5 Evaluation

In the previous section, we have presented two approaches to determine the footprint of an operation using a simple example to illustrate them. In this section, we report on experiments and case studies designed to evaluate our work. First, establishing footprints of model operation only makes sense if there exists a class of operations whose model usage is not 100%, that is, whose footprints do not cover the model completely. We investigated the existence of such operations in a case study (Section 3.5.3). Static footprinting is then only usable when it is (a) valid, (b) precise and (c) efficient. We evaluated static footprinting along these 3 directions with some experiments in Sections 3.5.4 and following. Finally, we discuss the threats to the validity of our evaluation in Section 3.5.7.

3.5.1 Implementation

To evaluate our approach, we have chosen Kermeta [MFJ05] as the language for the definition of operations. This choice is accidental to our contribution; the notion of footprint is not bound to a particular

technical space [Béz05] or technology to implement model operations. Actually, estimating footprints may have been easier for operations defined with graph grammars [VVP02]. Nevertheless, the imperative and object-oriented nature of Kermeta makes it more accessible, and thus more relevant, to industrial practice. Furthermore, Kermeta is an executable metamodeling language compatible with the Eclipse Modeling Framework[1] (EMF), a popular implementation of EMOF based on Eclipse.

We have implemented both dynamic and static footprinting for operations written in Kermeta. For dynamic footprinting, we have extended the interpreter of Kermeta to save execution traces and have written a Java program to analyze them. The static analysis of Kermeta code has been implemented in Kermeta as a visitor on (type checked) abstract syntax trees of Kermeta programs. The filtering of models has been implemented in Java using the reflective API provided by EMF. Our implementation supports any input metamodels (examples include the metamodel for Petri nets in Figure 3.3, UML or any Domain Specific Language), as long as these metamodels are defined in Kermeta or in Ecore, which is the metamodeling language used in EMF. Details about our algorithms and their implementation may be found in [JGB11b].

[1] `http://www.eclipse.org/emf`

3.5.2 Models and Operations

The unit of analysis in our empirical work is the execution of an operation on an input model. For this evaluation, the models studied are actually metamodels written in Ecore. We decided to use metamodels instead of models of systems (*i.e.*, we use M2 models instead of M1 models in the OMG's four layered metamodel architecture [OMG11a]) because we had not enough real-world models at our disposal whereas there are many real-world metamodels available for researchers in the public domain. We have randomly selected a sample of 75 Ecore metamodels that were packaged in Eclipse plugins or available in online repositories, such as the *AtlanMod* metamodel zoo[2]. We list these metamodels in [JGB11b].

Furthermore, we have defined 5 model operations to be executed on (meta)models representing their usage. Since metamodels describe modeling languages, their typical use includes the documentation of modeling languages and storage and manipulation of models expressed in the modeling languages.

E2KV generates Kermeta code implementing a visitor for the input metamodel. This visitor can be used to implement operations on models conforming to the input metamodel.

E2SQL creates a SQL schema for storing, in a database, models expressed in the input metamodel.

[2] `http://www.emn.fr/z-info/atlanmod/index.php/Zoos`

E2HTML creates an HTML document presenting the metamodel (as does Javadoc for Java code).

E2DOT visualizes the input metamodel with the help of GraphViz[3], a tool for rendering graphs.

E2GEN generates a generator model of the metamodel, which contains additional information for code generation. This generator model decorates the input model, that is, it contains reference to the input model.

More details about these operations can be found in [JGB11b], including their source code and their static metamodel footprint. In addition, we consider the operation combining these 5 operations to illustrate footprints left by a set of operations:

E2* Suite executes sequentially E2KV, E2SQL, E2HTML, E2GEN and E2DOT.

In total, our evaluation is based on 450 executions of operations (5+1 operations and 75 models). We have produced dynamic and static footprints for these 450 executions.

3.5.3 Model Usage

Table 3.2 presents the average (median) model usage while Figure 3.7 displays the usage ratio measured in our sample of models with respect to the operations (based on dynamic footprints). Every point

[3] `http://www.graphviz.org`

Table 3.2: Average model usage per operation.

	E2KV	E2SQL	E2HTML	E2DOT	E2GEN	E2* Suite
η_o	16.58%	49.43%	55.56%	51.70%	51.70%	55.86%
η_s	4.46%	25.43%	21.47%	22.57%	8.38%	39.25%

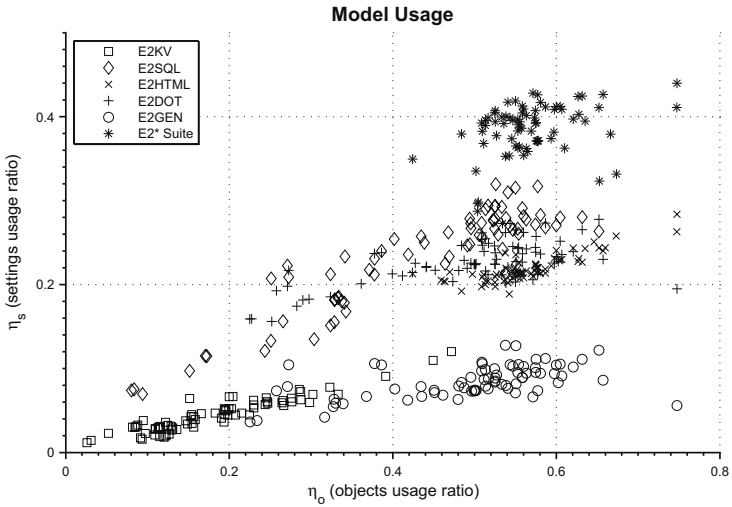


Figure 3.7: Model Usage of 75 models by 6 operations.

represents the usage of one model by one operation. On the horizontal axis, we measure the usage in terms of objects (η_o), while the usage in terms of settings (η_s) is measured on the vertical axis. The plot is to be interpreted as follows: the higher / more right a point, the more an operation uses the elements of a model (in terms of settings respectively in terms of objects).

For example, when measuring the representation of UML in Ecore

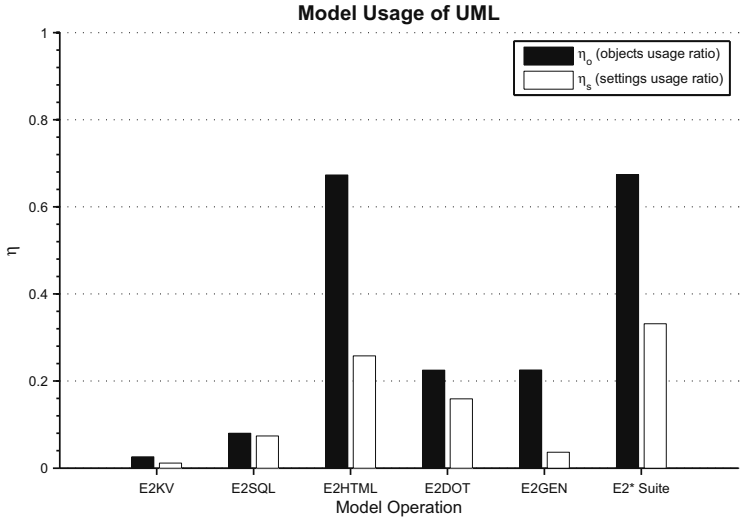


Figure 3.8: Usage of UML by 6 operations.

(Figure 3.8), we found that only 3% of its objects and 1% of its settings are used when creating a visitor in Kermeta for UML (E2KV). Note that many of the unused elements are used by other operations. Indeed, with respect to E2HTML, the model usage of UML increases to 67% in terms of objects and to 26% in terms of settings.

In average (median), 56% of the objects and 36% of the settings in a model are used with respect to our operation suite E2* (see Table 3.2). The low usage of E2KV can be explained by the fact that this operation is only interested in `EClass` objects and the inheritance relationships among them but not their content (such as `EAttribute` or `EOperation` objects). On the opposite, E2HTML considers almost

every kind of model element (including some of their annotations). Nevertheless, we found no footprint that completely covers a model, because none of our operation uses `EGenericType` objects.

3.5.4 Validity of Static Footprints

To be of any use, static footprints must be conservative estimates of dynamic footprints. This property should hold by construction, but we nevertheless verified that our implementation satisfies this requirement by testing it with 450 test cases (1 test case per footprint). We executed the operations on both static footprints and complete models and compared the outcome of these executions. We found no difference between the outputs of these executions. In other words, all elements needed by the operations were indeed included in the static footprints. Note that E2GEN (and, consequently, E2*) created slightly different outputs, because the operation creates decorator models containing references to the input models. Thus, when E2GEN is executed on a static footprint, the output model refers to the static footprint rather than the complete model. This difference is insignificant and does not argue against the validity of static footprints.

3.5.5 Precision of Static Footprints

Since static footprints estimate dynamic footprints, we can assess the pertinence of this estimation by using the precision measure from the

information retrieval field. For this purpose, we consider elements of the dynamic footprint as *relevant* while elements from the static footprint form the set of *retrieved* elements. *Precision* measures the proportion of retrieved elements that are indeed relevant. Note that the dual of precision, *recall* (the proportion of relevant statements that have been retrieved), is always trivial in this context (100%), because a static footprint is always a superset of the dynamic footprint it estimates. The precision of the estimation is measured by considering objects (σ_o) and settings (σ_s). The larger the measure σ, the closer is the static footprint to the dynamic footprint and the more accurate is the measure of model usage when it is based on static footprinting ($\widehat{\eta}$).

$$\sigma_o = \frac{\text{\# objects in dynamic footprint}}{\text{\# objects in static footprint}}$$
$$\sigma_s = \frac{\text{\# settings in dynamic footprint}}{\text{\# settings in static footprint}}$$

In our explanatory example (see Section 3.4), both footprints contain 10 objects. In terms of settings, the dynamic footprint has 21 settings while the static footprint contains 26 settings. Therefore, the precision of the static footprint is $\sigma_o = 1$ and $\sigma_s = 0.81$.

Figure 3.9 presents the precision of our 450 static footprints in comparison with their dynamic counterparts. On the horizontal axis, we

Table 3.3: Average precision of static footprints.

	E2KV	E2SQL	E2HTML	E2DOT	E2GEN	E2* Suite
σ_o	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
σ_s	89.26%	92.48%	92.89%	95.80%	65.71%	94.12%

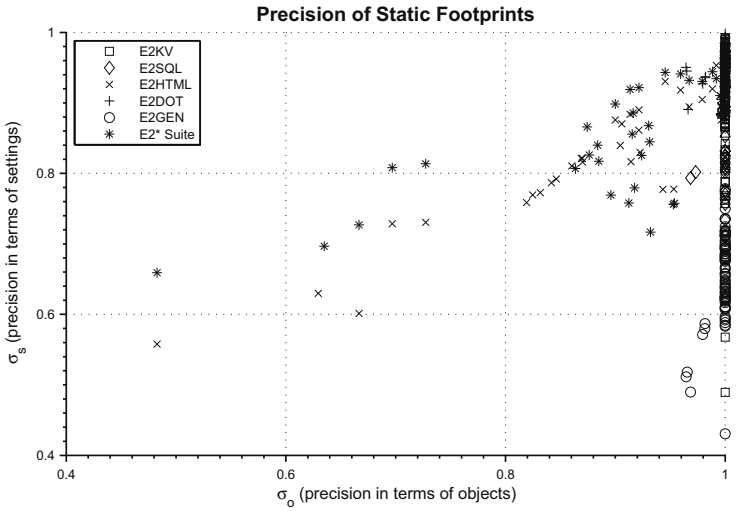


Figure 3.9: Precision of static footprints with respect to dynamic footprints.

measure the precision in terms of objects (σ_o), while the precision in terms of settings (σ_s) is measured on the vertical axis. Therefore, the higher / more right a point, the closer is the static footprint to the dynamic one.

In average, static footprints are very precise: the majority of static footprints contain no irrelevant objects (the median of σ_o is 100%) while still containing some irrelevant settings (see Table 3.3).

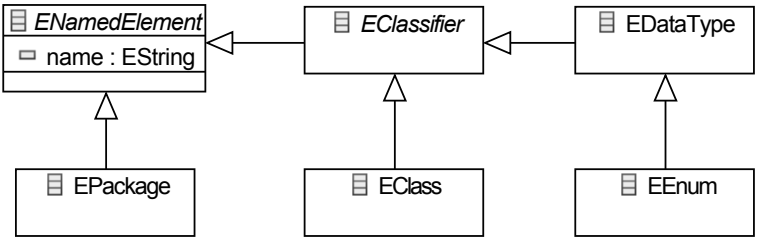


Figure 3.10: Excerpt of the Ecore metamodel: ENamedElement and some of its subtypes.

On the left part of the plot in Figure 3.9, there are 10 static footprints with a precision $\sigma_o < 0.8$. These are the footprints of five models (friends, BPMN, filesystem, flowchart and fsmStatic) with respect to two operations (E2HTML and E2*). The imprecision of these static footprints is due to `EAnnotation` objects. E2HTML (and consequently E2*) reads annotations whose *source* is "genmodel". These five models have a lot of annotations, but these annotations have different sources (they are destinated to other operations or relevant to other purposes). Dynamic footprints do not include the entries of these `EAnnotation` objects while static footprints include them. These outliers reveal a limitation of static footprinting: its precision can be rather low for operations whose usage depends on conditions defined in terms of object states.

Many static footprints suffers from a lack of precision in terms of settings $\sigma_s < 0.6$ (lower left part of Figure 3.9). This lack of precision is due to the inheritance relationships among the types of the Ecore

metamodel. An excerpt of this metametamodel is depicted in Figure 3.10: The feature *name* is defined in a class `ENamedElement`, which is the supertype of many other classes.

Two static footprints of E2KV are impacted by this problem. The static metamodel footprint of E2KV contains all classes depicted in Figure 3.10. Still, the operation only reads the *name* of `EClass` objects, but not the *name* of `EDataType` objects. Since Ecore and BPMN contain a lot of `EDataType` objects, their static footprints get imprecise in terms of settings, because these static footprints include settings for the name of `EDataType` objects, while the dynamic footprints do not include them.

E2GEN further illustrates this problem. Its static metamodel footprint also contains all classes of Figure 3.10. However, E2GEN only reads the *name* of `EPackage` objects. Thus, many objects in the static model footprints of E2GEN will have a setting for *name*, while only `EPackage` objects will have a setting for it in the dynamic footprint. This explains the low average of σ_s for E2GEN in Table 3.3.

Other operations read the *name* of each `ENamedElement` object in their static footprints. Thus, their static footprints are not impacted by this issue.

Table 3.4: Computation time of footprints.

Model	# Objects	Method	E2KV	E2SQL	E2HTML	E2DOT	E2GEN	E2* Suite
DocBook	23	Dynamic	1'955 ms	1'587 ms	1'525 ms	1'615 ms	1'541 ms	3'193 ms
		Static	8 ms	11 ms	10 ms	10 ms	10 ms	12 ms
		SpeedUp	*244x*	*144x*	*153x*	*162x*	*154x*	*266x*
XQuery	100	Dynamic	1'832 ms	1'782 ms	1'792 ms	2'171 ms	1'637 ms	3'863 ms
		Static	16 ms	24 ms	23 ms	24 ms	22 ms	22 ms
		SpeedUp	*115x*	*74x*	*78x*	*90x*	*74x*	*176x*
XHTML	1'035	Dynamic	3'211 ms	6'460 ms	8'762 ms	4'262 ms	2'238 ms	20'495 ms
		Static	101 ms	132 ms	108 ms	90 ms	66 ms	59 ms
		SpeedUp	*32x*	*49x*	*81x*	*47x*	*34x*	*347x*
OCLUML	13'849	Dynamic	6'348 ms	12'237 ms	41'267 ms	24'417 ms	5'426 ms	90'915 ms
		Static	222 ms	344 ms	651 ms	373 ms	111 ms	240 ms
		SpeedUp	*29x*	*36x*	*63x*	*65x*	*49x*	*379x*
Static Metamodel Footprint			6'242 ms	6'733 ms	6'878 ms	6'860 ms	6'910 ms	15'524 ms

3.5.6 Efficiency of Static Footprinting

In this subsection, we evaluate the cost of static footprinting and compare it to the cost of dynamic footprinting. Table 3.4 summarizes the results of this evaluation. We selected four models from our sample, each one representing an order of magnitude in terms of model size: the smallest ($\sim$ 10 objects), the largest ($\sim 10'000$ objects) and two in-between ($\sim$ 100 and $\sim 1'000$ objects). For dynamic footprinting, we measure the time needed for both executing the operation while keeping a trace of its execution and extracting the dynamic footprint out of this trace. On the opposite, static footprinting has an initial cost for analyzing the operation definition to extract its static metamodel footprint. Once this metamodel footprint has been computed, it can be used to filter any model. Thus, we keep the cost of precomputing the static metamodel footprint separated from the cost of filtering models.

When the static metamodel footprint is provided, static footprint (filtering a model) is always cheaper than computing dynamic footprints,

even for our smallest model and our simplest operation. The speed up ranges from 29× to 379×. When the metamodel footprint is not precomputed, static footprinting (precomputing the metamodel footprint and filtering a model) is faster than dynamic footprinting in 6 cases highlighted with shaded cells.

Execution times were measured as follows. Each footprint (whether dynamic, static model or static metamodel footprint) was computed 5 times, each time in a freshly started Java virtual machine. Table 3.4 displays the medians of these experiments. Kermeta code — model operations and the static analysis of model operations — was interpreted (and not compiled to Java). To minimize the influence of Eclipse internal mechanisms on our measure, we run Eclipse in headless mode and we forced the loading of required plugins by computing a dummy footprint before the one we were interested in. The computer used for this evaluation is an Intel(R) Core(TM) i7 @ 2.8 GHz with 8 Gb RAM running Windows 7 Professional, Java(TM) 1.6.0_20.b02 (64 bits) and Eclipse 3.5.2 with Kermeta 1.3.2.

3.5.7 Threats to Validity

We have only evaluated our approach with operations written by us. Thus, the evaluation of our approach may not be reliable and results may differ with a different set of operations. Our experiments require both models and operations. If we had used published model operations (such as those presented in [CM00] or [KAER06]), we

would have had to create UML models. Creating models rather than model operations would have been an equally strong threat to the validity of our experiments. Still, to mitigate this threat, we either based our operations on existing tools (E2DOT, E2GEN and E2HTML) or benchmarks from the MDE community [BHRV08] (E2SQL) or used their results for implementing our approach (E2KV). Furthermore, the source code of these operations can be found in [JGB11b].

Finally, we only considered Ecore metamodels and operations written in Kermeta, because of the availability of many metamodels written in Ecore. There is no apparent reason to believe that static footprinting would be less precise or less efficient in other settings (*e.g.*, models of software systems expressed in UML), but the limited scope of our experiment remains nevertheless a threat to its external validity.

3.6 Discussion

Measuring model usage and footprinting are meaningful, since many operations do not use all the information contained in the input models. Still, we did not investigate empirically to which extent model usage or footprinting improve modeling processes or increase the quality of models. This is future work.

Because a footprint contains all elements touched during the execution of an operation, the operation produces the same output when

executed on a footprint as if it were executed on the complete model. There are nevertheless some exceptions. When an operation modifies its input models in place, such as refactorings, "untouched" model elements are implicitly copied to the output model and therefore impact the outcome of the operation. Thus, these "untouched" elements should be part of model footprints, but are not included by the algorithms presented in this paper. A similar issue exists with operations creating decorator models like E2GEN, that is, when the output model contains references to the input model. In these cases, footprints cannot be used in place of complete models as input to operations.

Furthermore, if an operation is not deterministic, *e.g.*, it uses collections such as sets or hashtables, the outcome of the operation may differ when it is executed on a footprint or on the complete model. Still, the differences are often meaningless, *e.g.*, the ordering of classes in a package.

Static footprinting estimates a dynamic footprint by considering types only. Therefore, static footprints become imprecise when an operation definition involves classes that have many subclasses. In addition, static footprinting ignores conditions expressed in terms of objects states. For example, the footprint of an operation working only on a `EPackage` called "persistence" will produce a static footprint containing all packages. In such situations, static footprints cannot be used for diagnosing model scope issues. Improving the precision

of static footprints by using a more sophisticated static analysis and filters than presented in this paper is left for future work.

In a similar direction, if a model operation relies on a reflection mechanism (*e.g.*, an interpreter which, given an OCL constraint and an UML model, evaluates the constraint on the UML model), the static footprint will be the complete model, as it is impossible to infer, from the model operation definition only, which objects or settings will be touched during the execution of the operation. In this case, one must resort to dynamic footprinting.

So far, we have implemented our approach only for operations written in Kermeta, because its imperative and object-oriented nature makes it more accessible to industrial practices. We could implement footprinting for other transformation languages such as QVT [OMG11b] or VIATRA [VVP02]. Actually, computing the static metamodel footprint of an operation written as a graph transformation is almost trivial: it suffices to collect metamodel elements present in the left-hand side argument of each transformation rule. Furthermore, some transformation languages have dedicated support for tracing a transformation's execution [CH06], making the dynamic footprinting approach easy to implement.

In this paper, we focus on models whose purpose is to feed one or more model operations. However, such models may contain information (*e.g.*, comments) that only serves for better understanding by humans and is not used in any model operation. When naively

computing the footprint of a set of model operations, the result will indicate "underuse" of the model, which actually is not true. In this situation, it is important to analyze not just the absolute size or model coverage of footprints, but compare the actual footprint to the expected one.

3.7 Related Work

Many frameworks have been proposed to define and evaluate the quality of models. Among others, a good model is at the right level of detail [DOJ+93] for its purpose. For Lindland [LSS94], a model is semantically correct if it contains all statements that are correct and relevant for the problem at hand (completeness), but nothing more (validity). In [SR98], Schuette and Rotthowe propose the minimalism criteria to operationalize their principle of construct adequacy. A model is *minimal* if none of its elements can be removed without a loss of information for the potential model users. These criteria rely on the evaluation of the relevance of the content of the model to the problem at hand, but, to the best of our knowledge, no objective measures has yet been proposed to quantify this attribute. As Davis *et al.* point out [DOJ+93], this attribute is difficult to measure because it is highly scenario-dependent. Since more and more of activities involving models become automated (especially in a MDE environment [MA07]), we propose to use model operations for characterizing the purpose of a model. Thus, footprints can be

used to define and measure the relevance of model elements based on their usage by model operations.

Along this line, footprints are best used with editors designed to visualize a single underlying model from various viewpoints, such as the Adora editor [GBJ02] or the orthographic modeling environment [AS08]. Their visualization techniques can hide model elements irrelevant for a given model operation, producing a view specifically tailored for it.

Formally, footprinting is a form of model slicing. However, unlike other model slices (such as [KSTV03]), our slicing criterion is not defined in terms of the behavior depicted by the model being sliced, but is related to the behavior of a model operation executed on the model.

Furthermore, model footprints can be used for impact analysis, *i.e.*, deciding whether a change in a model impacts the outcome of the operation. In [Egy06] for example, Egyed uses dynamic footprints of consistency rule instances (these footprints are called *scope* in [Egy06]) to decide whether a given rule instance (that is, a rule evaluated with respect to a given model element) must be reevaluated after a change in the model. Later, he extended his technique to generate fixes for inconsistencies [Egy07]: Since the scope of a rule instance contains all elements that affect its truth-value, at least one of these elements must change to fix the inconsistency. With these papers, Egyed

demonstrated the advantages of instance-based incremental consistency checking over type-based incremental consistency checking. In our work, we are interested in operations applied to the model in its entirety, which typically requires more time to execute than verifying some conditions on some set of elements. Thus, we are interested in estimating footprints statically, rather than tracing the execution of the operation.

A metamodel footprint documents the usage of an operation. Many modeling methods prescribe which kind of details is to be modeled for a given perspective (*e.g.*, [Kru95] or [RW05]). Sometimes, the creators of operations document explicitly which elements their operation uses (*e.g.*, Section 3 of [CM00]). Static footprinting not only generates automatically this documentation from the definition of operations, it can also be used to identify model elements that are excessive according to this documentation.

Metamodel footprints can also be used during the validation of model operations by checking that the metamodel footprint covers all relevant modeling constructs. [KAER06] lists metamodel coverage as a possible fault in model transformations. This motivated Wang *et al.* to propose a tool analyzing the metamodel coverage of model transformations [WKC06]. In contrast to their analysis, static metamodel footprints can be computed from model operations written in imperative languages.

Finally, metamodel pruning [SMBJ09] is a generic algorithm which, given a metamodel and a set of its elements, extracts a view from the

metamodel containing all these elements. In our work, we extract footprints from models. We use a variation of this algorithm to extract a view of static metamodel footprint from the complete metamodel (for an example of such a view in this paper, see Figure 3.6).

3.8 Conclusion and Future Work

During its execution, a model operation typically uses only a fraction of the content of its input models. The part of a model actually touched by a model operation forms its footprint. Measuring the model usage of operations (comparing the size of a footprint with respect to the size of a model) helps project managers to detect problems within the models and the operations involved in their project. Then, establishing the footprint of some operations with respect to a model supports modelers in reducing the scope and decreasing the level of details in the model so that it only contains the information that impacts the outcome of these operations. Moreover, footprinting gives hint for finding defects in operations and may suggest improvements for them.

In this paper, we have presented a method to reveal the footprint left by a model operation (dynamic footprinting) and a method to estimate this footprint without executing the operation (static footprinting). We have implemented both approaches and compared them on 75 models and 5 model operations (+1 combining these

operations). This experiment suggests that static footprinting can estimate dynamic (actual) footprints with a high precision (in average, 100% in terms of objects and 94% in terms of settings for E2*). Furthermore, static footprinting is between 29 and 379 times faster than dynamic footprinting when the static metamodel footprint of an operation is precomputed.

We believe that footprinting will reveal helpful information to build models at the right level of detail for their purposes. This suggests two complementary directions for future work. So far, we only considered the part of a model that was actually used and ignored the missing elements in a model that could have been used by an operation if they were modeled. Using static metamodel footprints, it may be possible to provide modelers with some hints on missing elements in their models. Furthermore, automated model operations are only a fraction of uses of models. We plan to investigate means to assess objectively the adequacy of a model's level of detail with respect to other modeling purposes.

Chapter 4

Analyzing Model Quality with Metamodel Footprints

Original publication:
Analyzing the Quality of Models and Model Operations with Metamodel Footprints
C. Jeanneret, M. Glinz, B. Baudry and B. Combemale
Submitted to SoSyM for publication

Abstract

In a Model Driven Engineering context, engineers create models, metamodels and model operations to represent software systems and to reason about them. Analyzing which parts of a model or metamodel are actually used by model operations may reveal quality problems in these artefacts,

such as models at the wrong level of abstraction or erroneous model operations. As these artefacts can be very large and complex, engineers may encounter problems to keep track of the footprint of an operation, that is, the set of elements it actually uses. In this paper, we present how the footprint of an operation can be computed, visualized and used for validating models and operations. We demonstrate the usefulness of footprints in various experiments involving 75 models, five model operations and four MDE experts. Footprints can help engineers in improving their models and operations or, otherwise, increase their confidence in the quality of their artefacts.

4.1 Introduction

To deal with the complexity of software, engineers use models to reason about it or its context. Models allow them to focus on important issues while ignoring irrelevant details, reducing the cognitive load imposed by the task at hand. The reasoning takes the form of model operations performed on models, such as queries, analyses, simulations or even transformations to new artefacts. In Model-Driven Engineering (MDE), many model operations are automated. This improves the reliability of the operations and reduces the time needed for their performance. To support the execution of such operations, models must be expressed in a language with a formal syntax. This syntax is often specified as object-oriented metamodels. Metamodels play an important role in any modeling activity, because they define an interface between the modelers and the users of the models.

When performed on a model m, an operation op computes new information based on the information stored in the model. During its execution, it navigates through the content of the model and gathers some information by reading some of its elements. The set of elements touched by a model operation during its execution forms the *footprint* m_{op} of that operation (see Figure 4.1). Thus, the footprint contains all elements that affect the outcome of the operation, as long as this operation is deterministic and does not use data other than those contained in the input model.

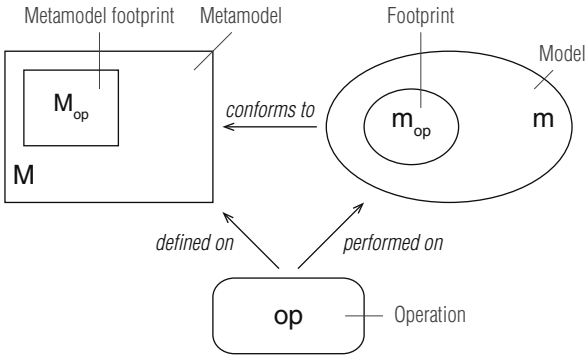


Figure 4.1: Footprints and metamodel footprints.

Frequently, model operations concentrate on certain kinds of model elements and do not touch the rest of the model. In metamodeling terms, this means that not all elements in the metamodel M will be relevant for an operation. When creating a model, modelers often ask themselves: "Have I modeled everything needed for that transformation?" or "Does this metaattribute matter for this analysis?" Other experts may ask similar questions when writing model operations: "Should I ignore this metaclass in my query?" or "Can I reuse this operation for this new metamodel, which is similar to the old one?" To answer these questions, the experts need to know the *metamodel footprint* M_{op} of the model operation, that is, the set of modeling constructs used by this operation (see Figure 4.1).

It is worth to compare the actual footprint of an operation with its expected or specified one. Indeed, there may be differences between

what an operation is supposed to use, what it is believed to use and what it actually uses. These differences may lead modelers to create models at the wrong level of abstraction or with the wrong scope or models that are incomplete for the operations they enable. These differences may also be explained by incomplete or erroneous operations. Thus, computing and comparing footprints is a method to detect such issues, increasing the confidence of modelers that they have modeled the right things and the confidence of engineers that their operations are using the right things.

Most of the time, engineers compute and reason about the footprint of an operation mentally and implicitly. However, this is a difficult exercise when the operation and the metamodel are large and complex. For example, the metamodel of UML, a standard modeling language, defines 255 types and 594 structural features[1] (such as attributes or references) and is so complex to use and to maintain that it has received the name of "metamuddle" in the literature [FGDTS06]. It may also happen that the source code of the operation is not available to the modelers or that this source code is written in a language they do not understand. Thus, tool support is required to extract and visualize the footprints of operations. Besides, explicit footprints have the benefit to enable the discussion among stakeholders.

In this article, we present an approach to compute and visualize metamodel footprints. Our approach is built on top of the Kompren

[1]The size of UML was measured on its implementation for Eclipse (v4.0.0) `http://www.eclipse.org/uml2/`

framework [BCBB12], which is a DSL for specifying and generating model slicers. We also present static footprinting, where metamodel footprints are used to estimate the footprints of operations when these operations are executed on models. We then demonstrate the benefits of metamodel footprints for validating both models and operations. For the first scenario, we conducted some experiments, showing that static footprinting is a precise yet efficient technique to estimate footprints. For the second scenario, we performed a case study where four MDE experts use metamodel footprints to locate defects in a model operation. These results highlight the effectiveness of metamodel footprints in these scenarios.

This article extends [JGB11a], which presents and compares two methods to compute the footprint of an operation: dynamic footprinting and static footprinting. Dynamic footprinting reveals the actual footprint of an operation after its execution, while static footprinting estimates the footprint by filtering the model through the metamodel footprint of the operation. This article rather focuses on metamodel footprints, a by-product of static footprinting. It augments the results presented in [JGB11a] by evaluating the use of metamodel footprints to validate model operations. Besides, we update our implementation to use the Kompren framework [BCBB12], which provides tool support for extracting and visualizing model slices.

The remainder of this article is organized as follows. In Section 4.2, we motivate our approach by presenting its main applications. Then

we present how metamodel footprints can be computed (Section 4.3) and visualized (Section 4.4). In Section 4.5, we explain how model footprint can be estimated with the help of metamodel footprints. We evaluate the approach in Section 4.6 and discuss the results in Section 4.7. Finally, we present related work (Section 4.8) before concluding the article in Section 4.9.

4.2 Motivation

4.2.1 The Importance of Metamodel Footprints

Usually, a modeling assignment consists of an original to be modeled, a language in which the model must be expressed and a modeling purpose. When creating a model, the modeler elicits information from a domain expert — who knows about the original — and expresses this information with statements in the modeling language. During this process, some information about the original is abstracted: only statements relevant for the purpose at hand are kept in the model.

Modeling is valuable on its own: it makes explicit some knowledge that may only exist in the expert's mind, which enables cooperation among stakeholders. However, the essential value of a model consists of using it as a substitute of the original to infer some new information about the original [Lud03]. As depicted in Figure 4.2, the

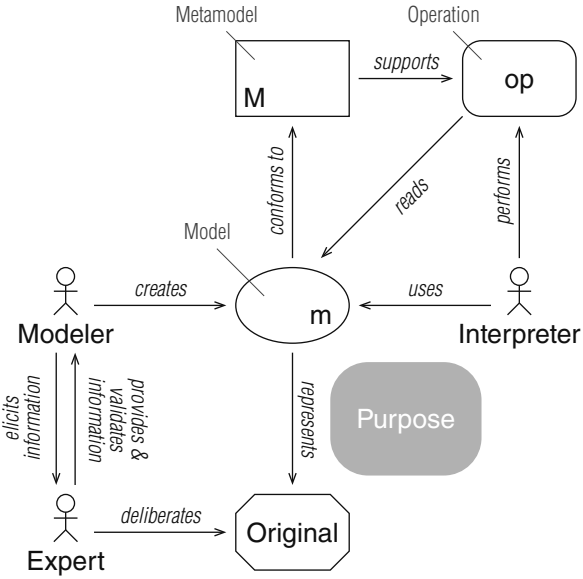


Figure 4.2: The roles and entities involved in a modeling activity.

inferences are made by the interpreter — the user of the model — by performing various model operations on the model, such as analyzing it, querying it or transforming it to other models or artefacts. In [JGB12], we propose to characterize the purpose of a model with the set of model operations to be performed on that model. These model operations can be performed mentally by a human interpreter or executed automatically by a computer. In either case, the model must conform to some structure, as defined by the modeling language. In this paper, we assume that the language is defined by an object-oriented metamodel.

In this setting, the metamodel is a key element, because it defines an interface between the modeler and the interpreter. Indeed, it describes which constructs can be used by the former to model the original and it provides the structure needed by the latter to define operations. However, defining a metamodel for each situation is not an option. It would prevent the reuse of models conforming to it and operations depending on it in other contexts. Furthermore, modelers and interpreters would have to learn a new language on a regular basis, impeding their productivity and proficiency. To avoid this, many engineers rely on UML, a standard general-purpose modeling language. To cover a wide spectrum of needs, UML is very large and complex. Besides, it can be extended for domain specific purposes. Yet, only a fraction of the language is typically used.

For example, consider a software engineer who wants to analyze the performance of a software system under construction. As the system

does not exist yet, she has to use a model of it. For this, she documents the architecture of her system according to the "4+1" viewpoints defined by Kruchten [Kru95]. Her model will include many UML diagrams: (1) use case and sequence diagrams for the scenario view, (2) a class diagram for the logical view, (3) a component diagram for the development view, (4) an activity diagram for the process view and (5) a deployment diagram for the physical view. She intends to use the approach proposed by Cortellessa [CM00] to convert her UML model into a performance model based on queueing network. This performance model will, in turn, be used to simulate the system and compute various performance metrics.

In fact, most model operations only use a subset of the constructs provided in a metamodel. The translation of UML models to performance models perfectly illustrates this, as this operation only uses constructs from use cases diagrams, sequence diagrams and deployment diagrams. We call this subset the metamodel footprint of the operation. If we have a set of operations, we define the metamodel footprint of this set as the union of the metamodel footprints of each individual operation (see Figure 4.3).

Once the metamodel footprint of a set of operations is known, the large metamodel can be pruned [SMBJ09], such that only constructs relevant for the operations are kept. Such a pruned metamodel is useful to engineers when they work on models and model operations, because they can concentrate on the constructs of interest without being distracted by the others. The metamodel footprint can also be

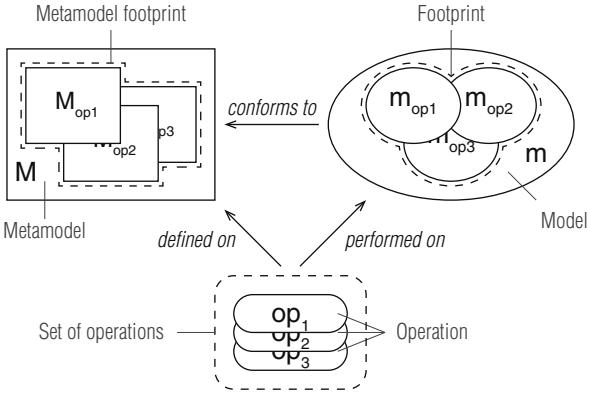


Figure 4.3: Footprints of multiple operations.

used to validate models and operations. In particular, the metamodel footprints allow computing the footprints of operations to identify excessive elements in models (Scenario 1). Metamodel footprints can also be used to suggest missing information in models (Scenario 2). Finally, it can be used to validate model operations by further constraining their input domains (Scenario 3).

4.2.2 Scenario 1: Validating Models

The footprint m_{op} of an operation is the set of all model elements touched by the operation during its execution. The footprint of a set of operations is the union of the footprints of each individual operation (see Figure 4.3). A model element that is not used by a set of

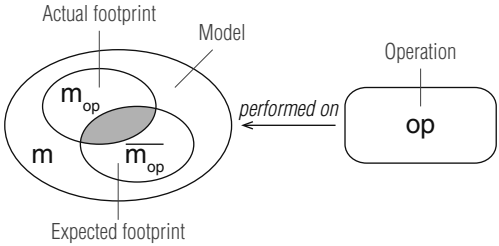


Figure 4.4: Validating models with footprints.

operations is excessive with respect to these operations, because such a model element does not impact the result of the operations. Models with excessive information may require more time and resources for their creation and their processing by these operations than necessary. More importantly, excessive elements may be due to serious problems in models or operations: (i) the models have the wrong scope, (ii) they contain information that nobody needs or they are on the wrong level of abstraction (*i.e.*, they contain unnecessary details), (iii) they serve more or other purposes than initially intended, or (iv) the operations concerned are erroneous or incomplete.

In the example given above, the presence of nodes in the deployment diagram that are not involved in any scenario (interaction) is a problem of scoping (i). Furthermore, the class diagram of the logical view is completely ignored by the operation (ii). Problem (iii) can be illustrated with the presence of comments meant to improve the

understandability of the model. Finally, the operation ignores elements from the activity diagrams. Still, these diagrams could provide useful information about workload derivation [CM00], suggesting a possible improvement to the operation (iv).

The modeler can identify information that is excessive with respect to an operation by establishing the footprint of this operation m_{op} and compare it with the expected footprint $\overline{m_{op}}$ (see Figure 4.4). Actual footprints are computed with dynamic footprinting. Dynamic footprinting executes the operation while tracing which elements it accesses. This method is expensive, because it requires executing the operation. To address this, we invented a technique called static footprinting that can estimate the footprint of an operation without executing it. The static footprint $\widehat{m_{op}}$ is computed by filtering the model m, keeping only those elements that are instances of constructs in the metamodel footprint M_{op}. The static footprint $\widehat{m_{op}}$ is a conservative estimate of the dynamic footprint m_{op} (see Figure 4.5). More details about dynamic and static footprinting will be given in Section 4.5.

The footprint of an operation contains all the information used for its execution. Thus, it can also be used for impact analysis. Any change that occurs in the footprint may have an impact on the outcome of the operation. Conversely, to improve the output of an operation, a modeler must focus on the elements of the footprint. Therefore, it is valuable to generate views from the model based on the footprints of the various operations it enables.

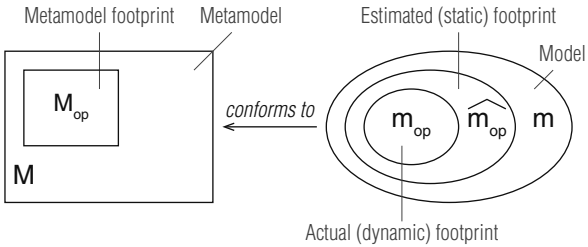


Figure 4.5: Estimating footprints with metamodel footprints.

4.2.3 Scenario 2: Completing Models

The metamodel footprint M_{op} is the set of all metamodeling constructs used by an operation *op*. By comparing the part M_m of the metamodel M used in the model m with the metamodel footprint M_{op} (see Figure 4.6), it is possible to identify missing elements in m. Ideally, M_m and M_{op} should be the same set. If a construct in M_{op} is not used in the model m, it may indicate that an element is missing in this model. Indeed, it means that the operation might have used this element if it were present. The model can be completed by creating elements instances of the missing constructs. In the example given above, the architect knows that her model completely covers the metamodel footprint of the operation, increasing her confidence in its completeness.

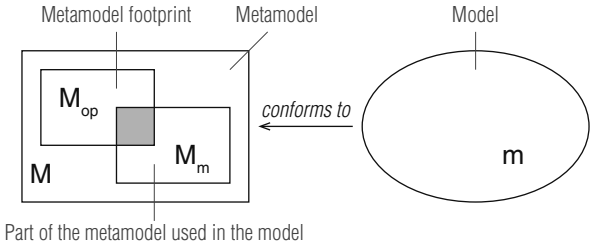


Figure 4.6: Completing models with the help of metamodel footprints.

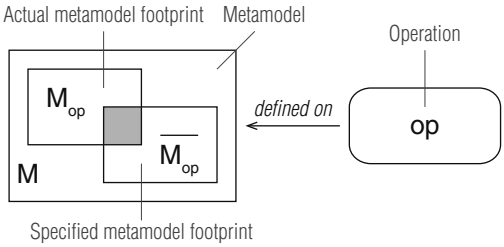


Figure 4.7: Validating operations with metamodel footprints.

4.2.4 Scenario 3: Validating Model Operations

A metamodel footprint can also be used as a specification for an operation. Indeed, it can be used to specify its input domain more precisely than just the complete metamodel. Then, it is possible to validate an operation by comparing the actual metamodel footprint M_{op} with the specified one $\overline{M_{op}}$ (see Figure 4.7). The differences between the two metamodel footprints can have one of the following explanations: it is (i) a specification error, (ii) an alternative implementation, (iii) a possible improvement of the operation or (iv) an implementation error.

Specification errors happen when some modeling constructs turn out to be needed for implementing an operation (thus, they are present in the actual metamodel footprint) but are not specified as such. Sometimes, there are several ways to implement an operation. Indeed, metamodels usually provide several ways to navigate among elements or propose various derived attributes to ease computations. In both cases, the specified metamodel footprint should be updated to match the actual metamodel footprint, so that these differences no longer exist in subsequent analysis.

The metamodel footprint of an operation $\overline{M_{op}}$ is specified by selecting constructs from the complete metamodel M. As this activity is done without the source code of the operation (or by somebody else than the implementer), one may notice some constructs that are

potentially useful, but not yet used by the operation. In the example given above, when visiting the UML metamodel, a specialist may consider the constructs used in activity diagrams as potential sources of information about workload derivation.

Finally, the implementation may be incomplete or erroneous. Suppose that the operation described in [CM00] is implemented, and that, unlike specified in Section 3 of [CM00], the actual metamodel footprint does not include any construct from the sequence diagrams. These differences between the specified and actual metamodel footprint help to find out which and how many constructs must still be dealt with by the operation.

Furthermore, a specified metamodel footprint defines a starting point for the implementation. For example, it specifies which constructs need to be visited if the operation is implemented with the visitor design pattern. In declarative implementations, it defines which constructs need to be matched in the operation. Even when the operation is already partially implemented, it allows the engineers to focus only on those constructs that are of interest for their operations.

4.3 Computing Metamodel Footprints

In this section, we present how to compute the metamodel footprint of an operation. To illustrate the further discussion, we introduce a running example. A simple Petri net model is given in

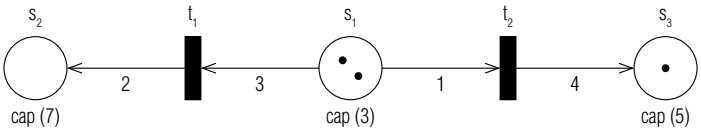


Figure 4.8: A Petri net.

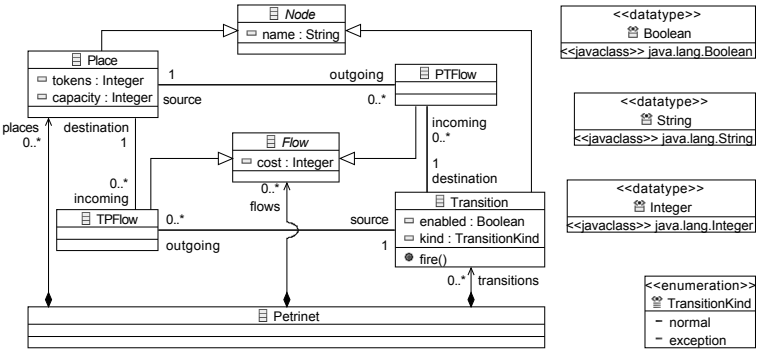


Figure 4.9: A metamodel for Petri nets.

Figure 4.8. Figure 4.9 shows a metamodel defining Petri nets of the kind given in Figure 4.8. Basically, such a *metamodel* is a set of modeling constructs: types and features. *Types* can either be (meta)classes or data types (enumerations or primitive types like integer). *Features* are divided into *structural features* (attributes within classes or references to other classes) and *behavioral features* (operations). For example, in the metamodel presented in Figure 4.9, `Transition` and `TransitionKind` are types, `capacity` and `source` are structural features and `fire()` is a behavioral feature.

In terms of its metamodel M, a *model* m can be regarded as a set of

model elements: objects and settings. Every *object* is an instance of a class defined in the metamodel. A *setting* represents a value or a set of values held by an object for a structural feature. Settings can be set to a given value, reset to the feature's default value or unset. The set of settings in an object forms the *state* of this object. In our Petri net example (Figure 4.8), place s_1 has the following five settings: (name = "s_1", tokens = 2, capacity = 3, incoming = $\emptyset$, outgoing = $\{s_1t_1, s_1t_2\}$).

We consider a query that extracts the names of enabled transitions in a Petri net. This operation can be formally defined in OCL [OMG12] as follows:

```
context Petrinet::namesOfEnabledTransitions():
    Set(String)
  body:
    self.transitions->select(t: Transition |
        t.enabled)->collect(t: Transition | t.name)
```

The attribute `enabled` (see metaclass `Transition` in Figure 4.9) is a structural feature whose value is derived from other values. A transition is enabled if (a) there are enough tokens in source places and (b) destination places have enough capacity to store additional tokens. This can be formally defined in OCL as follows:

```
context Transition::enabled: Boolean
  derive:
    self.incoming->forAll(f: PTFlow | f.source.tokens
        >= f.cost) and self.outgoing->forAll(f: TPFlow
        | f.destination.capacity - f.destination.tokens
        >= f.cost)
```

The *metamodel footprint* M_{op} of an operation *op* is the list of metamodel constructs — types and features — relevant for this operation. This list can be established by analyzing the definition of the operation. For operations written with declarative languages (such as triple graph grammar [Sch94]), the analysis is performed on the left-hand side pattern of each rule. For operations written in imperative languages (such as Java or Kermeta [MFJ05]), the analysis collects metamodel constructs along the control flow path, that is, the set of all its possible execution paths.

More precisely, for every expression involving a feature, we add the feature and the type of the object on which this feature is "applied" (accessed or invoked) to the metamodel footprint (unless these metamodel elements come from the language's library and not the input metamodel). For invocations of behavioral features (operations defined in classes), we also add the types of their parameters and their return types. For accesses to structural features, we include the type of this feature in the metamodel footprint.

Table 4.1 illustrates the analysis of the query from our running example. The left column lists all expressions found in its control flow graph that involve a feature from the Petri net metamodel. The second column lists the features involved in these expressions while the right column lists the types involved in the corresponding expression. Thus, the metamodel footprint of our example is the set of types and features in Table 4.1.

Table 4.1: Metamodel footprint of the query on Petri nets.

Expression	Feature	Types
self.transitions	*Petrinet::transitions*	Petrinet, Transition
t.enabled	*Transition::enabled*	Transition, Boolean
self.incoming	*Transition::incoming*	Transition, PTFlow
f.source	*PTFlow::source*	PTFlow, Place
f.source.tokens	*Place::tokens*	Place, Integer
f.cost	*Flow::cost*	PTFlow, Integer
self.outgoing	*Transition::outgoing*	Transition, TPFlow
f.destination	*TPFlow::destination*	TPFlow, Place
f.destination.capacity	*Place::capacity*	Place, Integer
f.destination	*TPFlow::destination*	TPFlow, Place
f.destination.tokens	*Place::tokens*	Place, Integer
t.name	*Node::name*	Transition, String

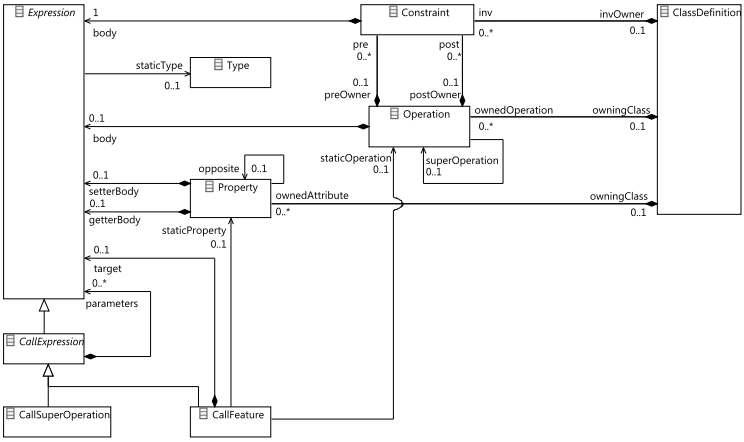


Figure 4.10: Excerpt of Kermeta's metamodel.

In our evaluation (see Section 4.6), we have chosen Kermeta [MFJ05] as the language for the definition of operations. Kermeta is an action language that extends EMOF [OMG11a] enabling the definition of model operations. Analyzing operations written in Kermeta is similar to the analysis of OCL expressions. Figure 4.10 displays an excerpt of Kermeta's abstract syntax. Among others, the metaclass `CallFeature` is of utmost interest, as it represents the access to an object's state or the invocation of an operation on an object. In Kermeta, `CallFeature` objects can be found in 3 contexts within the abstract syntax tree of a model operation:

- the body of a constraint (invariant of a class, contract of an operation)
- the body of an operation
- the body of the getter/setter of a derived property

We initially implemented the analysis to compute metamodel footprints as a visitor written in Kermeta accepting Kermeta abstract syntax trees [JGB11b]. As [BCBB12] points out, this analysis consists in slicing an abstract syntax tree to collect the types and features involved in it. Thus, this analysis can be formalized in Kompren, a DSL for defining modeling slicers [BCBB12]. Kompren tool support can generate a slicing function in Kermeta. This function visits the input model, starting from the input elements and following all sliced properties. During this visit, the function applies to all elements instances of the sliced classes the corresponding behavior.

```
slicer KermetaUsageAnalysis {
  domain: Kermeta.ecore
  input: Operation

  slicedClass: ClassDefinition cd [[M_op.add(cd)]]
  slicedClass: Enumeration en [[M_op.add(en)]]
  slicedClass: PrimitiveType pt [[M_op.add(pt)]]
  slicedClass: Operation op [[M_op.add(op)]]
  slicedClass: Property prop [[M_op.add(prop)]]

  slicedProperty: TypedElement.type
  slicedProperty: Operation.ownedParameter
  slicedProperty: Operation.body
  slicedProperty: Operation.pre
  slicedProperty: Operation.post
  slicedProperty: Operation.superOperation
      opposite(overridingOperations)

  slicedProperty: Expression.staticType
  slicedProperty: Block.statement
  slicedProperty: Block.rescueBlock
  slicedProperty: Conditional.condition
  slicedProperty: Conditional.elseBody
  slicedProperty: Conditional.thenBody
  slicedProperty: Loop.initialization
  slicedProperty: Loop.body
  slicedProperty: Loop.stopCondition
  slicedProperty: Raise.expression
  slicedProperty: Rescue.body
  slicedProperty: LambdaExpression.parameters
  slicedProperty: LambdaExpression.body
```

```
31    slicedProperty: LambdaParameter.type
32    slicedProperty: VariableDecl.initialization
33    slicedProperty: Assignment.target
34    slicedProperty: Assignment.value

36    slicedProperty: CallExpression.parameters
37    slicedProperty: CallFeature.target
38    slicedProperty: CallFeature.staticOperation
39    slicedProperty: CallFeature.staticProperty

41    helper [[
42    reference M_op: set NamedElement [0..*]
43    // M is a reference to the input metamodel
44    reference M: ModelingUnit
45    ]]

47    onEnd [[
48    M_op := M_op.select{e | M.contains(e)}
49    ]]
50  }
```

Listing 4.1: Computing the metamodel footprint of an operation written in Kermeta.

The analysis defines two references: the set M_{op}, which contains the metamodel footprint (line 42) and a reference to the input metamodel M (line 44). The analysis starts with an `Operation` object (line 3), which is the operation under analysis. It visits its return type (line 11), its parameters (line 12), its body (line 13), its contract (lines 14 and 15) and, if they exist, all overriding operations (line

16). As there is no metareference to overriding operations in the metamodel of Kermeta, we use the keyword `opposite` to create a metareference called `overridingOperations` that is the opposite of `superOperation`. When visiting an expression, the analysis visits its static type (line 18) and all its embedded sub-expressions (lines 19–34). When visiting a `CallFeature` object, the analysis visits the parameters (line 36), the target (line 37) and the referred operation (line 38) or property (line 39). During this visit, the analysis only collects the types (lines 5–7) and features (lines 8 and 9) into the metamodel footprint M_{op}. Once the slicing is complete, the analysis filters the metamodel footprint to only keep the elements part of the input metamodel M and to discard the ones part of the language's library (line 48).

4.4 Visualizing Metamodel Footprints

Formally, the metamodel footprint is a set of types and features. As such, it is only an abstract construct and it must be visualized to be of any use. A possible visualization is to extract a view from the input metamodel corresponding to the metamodel footprint. Such a view contains:

- the types contained in the metamodel footprint
- the features contained in the metamodel footprint

- the types owning features contained in the metamodel footprint
- the parameters of operations contained in the metamodel footprint
- the literals of enumerations contained in the metamodel footprint
- optionally, the subclasses of types contained in the metamodel footprint

Including the subclasses of types contained in the metamodel footprint can help to comprehend the metamodel footprint by making explicit all metaclasses used by the operation. However, it also increases the size and the complexity of the view extracted from the complete metamodel.

The following slicer, written in Kompren, extracts a view from the metamodel M according to a metamodel footprint M_{op}:

```
1  slicer strict ExtractMetamodelFootprint{
2    domain: Kermeta.ecore
3    input: NamedElement

5    slicedClass: NamedElement

7    slicedProperty: Property.owningClass
8    slicedProperty: Operation.owningClass
9    slicedProperty: Operation.ownedParameter
10   slicedProperty: Enumeration.ownedLiteral
11   slicedProperty: ClassDefinition.superType option
       opposite(subTypes)
```

```
12    slicedProperty: Class.classDefinition
13 }
```

Listing 4.2: View extraction from the input metamodel according to the metamodel footprint.

The keyword `strict` means that the generated slicing function behaves like a pruner: it produces slices that contain all the visited elements that are instances of sliced classes and, additionally, all the elements that are necessary to satisfy the structural constraints of the domain metamodel. In other words, this slicing function automatically includes some elements like `Package` objects that are required by the Kermeta metamodel even if they are not properly part of the slice. This slicer visits the input metamodel M, starting from the set of `NamedElement` objects (line 3) forming the metamodel footprint M_{op}. The slicing function collects all `NamedElement` objects it encounters during its navigation (line 5). It visits the elements mentioned above: the types owning features contained in the metamodel footprint (lines 7 and 8), the parameters of operations contained in the metamodel footprint (line 9) the literals of enumerations contained in the metamodel footprint (line 10), and, optionally, the subclasses of types contained in the metamodel footprint (lines 11 and 12).

Figure 4.11 presents a view of the Petri net metamodel (the complete metamodel is illustrated in Figure 4.9). This view contains all metamodel elements from Table 4.1, and, additionally, the classes

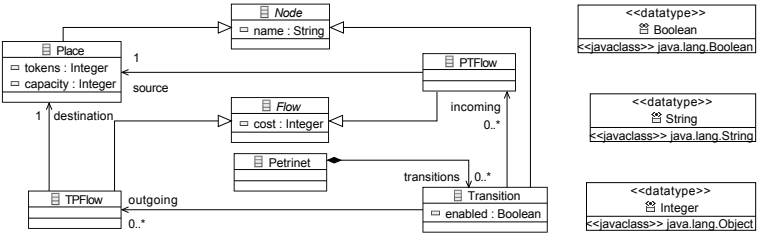


Figure 4.11: View from the Petri net metamodel based on the metamodel footprint.

`Node` and `Flow`. They have been included in this view, because some of their features — `name` and `cost` — are part of the metamodel footprint.

In most cases, the extracted view can display the inheritance relationships among selected classes. However, a problem arises when a class within an inheritance hierarchy is not selected because none of its proper features are used, but its supertypes and subtypes are. Then, the inheritance relationship between the subtypes and the supertypes are not displayed, because there is no explicit inheritance relationship between them in the metamodel M. The class hierarchy displayed in Figure 4.17 (page 149) can be used to illustrate the issue. If both `EClass` and `EModelElement` are selected, but not `EClassifier`, then the implicit inheritance relationship between `EClass` and `EModelElement` is not visible. This is problematic because it is impossible to infer that `EClass` objects inherit the `name` attribute from `EModelElement`. To address this issue, we create an explicit inheritance relationship between them, so that this rela-

tionship can be visualized. This solution is valid, because inheritance relationships are transitive.

4.5 Estimating Model Footprints

The footprint m_{op} of an operation is the set of model elements used by an operation during its execution. Actual footprints are calculated with dynamic footprinting. This technique consists of analyzing the execution trace of the model operation. Every time an operation is invoked on an object or every time a setting is read from an object, the target object, the setting and the (optional) parameters are added to the model footprint. When a setting refers to one or more objects, these objects are added to the footprint as well.

To illustrate this method, we consider the query that extracts the name of enabled transitions and the Petri net of Figure 4.8. The execution of that query on this model yields an execution trace consisting of 24 accesses to objects (assuming that OCL expressions are not evaluated lazily). An excerpt of this trace is shown on the left side of Figure 4.12, while the result of the trace analysis is presented on the right side. During its execution, the operation has touched 10 objects and 21 settings in total. The operation therefore uses all objects in the model but only half of its settings.

While dynamic footprinting can reveal the actual footprint of a model operation, it is expensive to perform because it requires executing the

#	object	feature
1	p: Petrinet	*transitions*
2	t1: Transition	*enabled*
3	t1: Transition	*incoming*
4	s1t1: PTFlow	*source*
5	s1: Place	*tokens*
6	s1t1: PTFlow	*cost*
7	t1: Transition	*outgoing*
8	t1s2: TPFlow	*destination*
9	s2: Place	*capacity*
10	t1s2: TPFlow	*destination*
11	s2: Place	*tokens*
12	t1s2: TPFlow	*cost*
13	t2: Transition	*enabled*
[…]		
24	t2: Transition	*name*

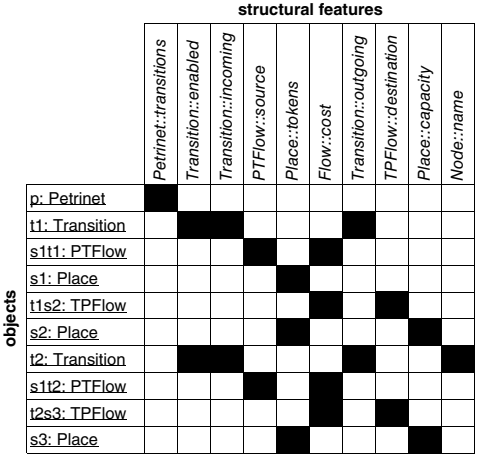


Figure 4.12: Dynamic footprinting of the Petri net example.

operation, tracing this execution and then analyzing this trace (see left part of Figure 4.13). In [JGB11a], we introduced static footprinting to estimate footprints. This method is divided in two steps (see right part of Figure 4.13). First, the metamodel footprint M_{op} is computed as explained in Section 4.3. Then, the static footprint $\widehat{m}_{op}$ is obtained by filtering the model: only those elements that are instances of the constructs of the metamodel footprint are kept.

Figure 4.13 illustrates both approaches. The major difference between them is that dynamic footprints can be obtained only *after* the execution of the operation, while static footprint can be computed *without* executing the operation. This difference has major impacts on both the effort required by these approaches and their precision.

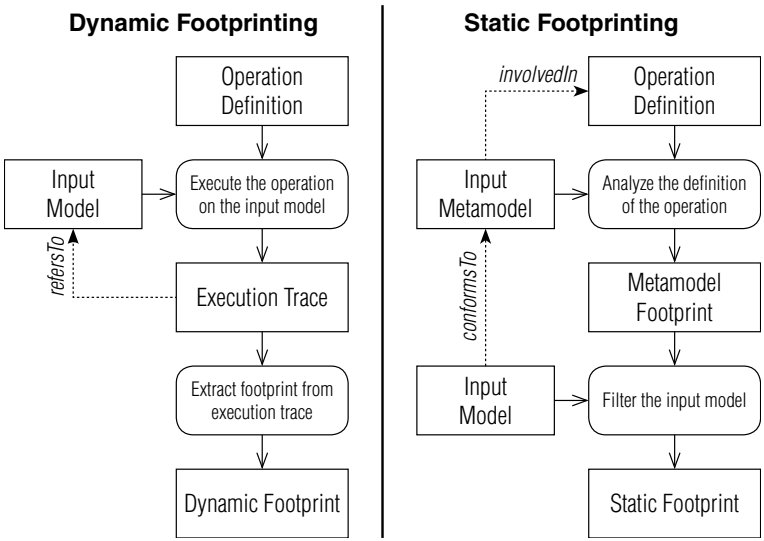


Figure 4.13: Dynamic and static footprinting.

We evaluate these differences quantitatively in Section 4.6.2 and we discuss their limitations in Section 4.7.

Computing the metamodel footprint M_{op} for a given operation has already been explained in Section 4.3. In [JGB11b], we explain how we implemented the filtering of a model m with respect to the metamodel footprint M_{op} in Java. Yet, such a filter can be expressed as a model slicer in Kompren. For this, a new slicer is generated from the metamodel footprint M_{op}. In our example, the metamodel footprint of Table 4.1 generates the following model slicer:

```
1 slicer Petrinet_4_PETN {
2   domain: Petrinet.ecore
```

```
3    input: Petrinet

5    slicedClass: Place p [[m_op.add(p)]]
6    slicedClass: Transition t [[m_op.add(t)]]
7    slicedClass: TPFlow t [[m_op.add(t)]]
8    slicedClass: Petrinet p [[m_op.add(p)]]
9    slicedClass: PTFlow p [[m_op.add(p)]]

11   slicedProperty: Place.tokens
12   slicedProperty: Place.capacity
13   slicedProperty: Transition.enabled
14   slicedProperty: Transition.outgoing
15   slicedProperty: Transition.incoming
16   slicedProperty: TPFlow.destination
17   slicedProperty: Node.name
18   slicedProperty: Petrinet.transitions
19   slicedProperty: PTFlow.source
20   slicedProperty: Flow.cost

22   helper [[
23   reference m_op: set Object [0..*]
24   ]]
25 }
```

Listing 4.3: Extract Footprint from Petri Net.

Its domain is the input metamodel and its input is the root classes of the metamodel, whose instances are containers for the model. It does not matter whether these root classes are included in the metamodel footprint. In our case, the metamodel is the metamodel for Petri nets (see Figure 4.9) which has only one root class: the

`Petrinet` class (line 3). Then, each class contained the metamodel footprint becomes a sliced class, while each structural feature becomes a sliced property. When the slicer visits an object that is instance of a sliced class, the slicing functions adds it to the footprint m_{op} (lines 5–9). The footprint is stored in the set m_{op} (line 23). Kompren automatically infers a metamodel for the footprint, containing all sliced classes and properties.

This slicer accepts a Petri net and extracts from it the model footprint corresponding to the query extracting the names of enabled transition. When applied to the Petri net in Figure 4.8, the slicer function returns 10 objects and 26 settings.

4.6 Evaluation

In the previous section, we explained how to derive and visualize the metamodel footprint of an operation. In this section, we report on experiments and case studies conducted to evaluate our approach. In Section 4.6.1, we present the design of our evaluation, which focuses on two scenarios: model validation (Section 4.6.2) and operation validation (Section 4.6.3). Finally, threats to validity are discussed in Section 4.6.4.

4.6.1 Evaluation Design

Research Goals and Questions

The goal of our evaluation is to demonstrate the usefulness of metamodel footprints in two scenarios presented in Section 4.2: estimating model footprints (Scenario 1) and validating model operations (Scenario 3). For the former scenario, we first investigate whether it makes sense to compute footprints by analyzing the average model usage of a set of model operations. Then we evaluate the validity, efficiency and precision of static footprinting with respect to dynamic footprinting. In this article, we do not evaluate how model footprinting helps modeler in creating models as the results of our attempts in this direction are inconclusive so far [JGBC12]. However, we conduct an end-to-end evaluation of our approach for the latter scenario.

Computing footprints only makes sense if models are not completely used by the operations they enable. Thus, we first investigate the average model usage of a set of operations on a set of models. The following exploratory question serves as motivation for model footprinting:

Question 1. *What is the average model usage of a set of operations?*

As we explained in Section 4.5, there are two footprinting techniques: dynamic and static footprinting. Dynamic footprinting calculates

the actual footprint, while static footprinting estimates it. Static footprinting is valuable only if it is more efficient than dynamic footprint and produces valid and precise estimates of footprints. Thus, we evaluate static footprinting on these three aspects with the following research questions:

Question 2. *Does static footprinting produce valid estimates of footprints?*

Question 3. *How precise is the estimation made by static footprinting?*

Question 4. *How efficient is static footprinting?*

The second scenario is the validation of operations. The evaluation of this scenario is end-to-end, that is, it involves MDE experts. In our approach, an operation is validated in two steps: First, its metamodel footprint is specified. Second, its actual metamodel footprint is compared with the specified one. Deltas are then analyzed and classified as problems or not. Our evaluation of this approach covers three aspects: its feasibility, the results it delivers and the effort it requires. This leads to the following research questions:

Question 5. *Can the metamodel footprint of an operation be specified before it is implemented?*

Question 6. *How many deltas are related to implementation errors?*

Question 7. *How long does it take to specify a metamodel footprint? How long does it take to compare the actual with the specified metamodel footprint?*

Models and Operations

To evaluate our approach, we need (a) an implementation of static and dynamic footprinting, (b) a set of models and (c) a set of operations. We have chosen Kermeta [MFJ05] as the language for the definition of operations. This choice is accidental to our contribution; the notion of footprint is not bound to a particular technical space [Béz05] or technology to implement model operations. Nevertheless, the imperative and object-oriented nature of Kermeta makes it more accessible, and thus more relevant, to industrial practice. Furthermore, Kermeta is an executable metamodeling language compatible with the Eclipse Modeling Framework[2] (EMF), a popular implementation of EMOF based on Eclipse.

For the evaluation, we have used models written in Ecore, which is the metamodeling language used in EMF. In other words, we used metamodels as models because we had not enough real-world models of systems at our disposal, while there are a lot of metamodels publicly available. This decision has no impact on the validity of our evaluation, as metamodels are just models with a different original (a modeling language instead of a system). We have randomly selected a sample of 75 Ecore metamodels that were packaged in Eclipse plugins or available in online repositories such as the AtlanMod metamodel zoo[3]. [JGB11b] lists these 75 models and their respective size.

[2] `http://www.eclipse.org/emf`
[3] `http://www.emn.fr/z-info/atlanmod/index.php/Zoos`

Furthermore, we have defined five model operations to be executed on (meta)models representing their usage. Since metamodels describe modeling languages, their typical use includes the documentation of modeling languages and storage and manipulation of models expressed in the modeling languages.

E2KV generates Kermeta code implementing a visitor for the input metamodel. This visitor can be used to implement operations on models conforming to the input metamodel.

E2SQL creates a SQL schema for storing, in a database, models expressed in the input metamodel.

E2HTML creates an HTML document presenting the metamodel (as does Javadoc for Java code).

E2DOT visualizes the input metamodel with the help of GraphViz[4], a tool for rendering graphs.

E2GEN generates a generator model of the metamodel, which contains additional information for code generation. This generator model decorates the input model, that is, it contains reference to the input model.

More details about these operations can be found in [JGB11b], including their source code and their metamodel footprint. In addition, we consider the operation combining these five operations to illustrate footprints left by a set of operations:

E2* Suite executes sequentially E2KV, E2SQL, E2HTML, E2GEN and E2DOT.

[4] `http://www.graphviz.org`

Table 4.2: Size of the metamodel footprint for each operation.

	E2KV	E2SQL	E2HTML	E2DOT	E2GEN	E2* Suite	Ecore
Types	7	11	21	16	14	21	52
Features	7	17	22	20	13	34	112

In total, Ecore defines 52 types and 112 features. As displayed in Table 4.2, the E2* operations set do not use the Ecore metamodel completely. Thus, computing the metamodel footprint of an operation makes sense because there is reasonable set of operations that does not use the whole metamodel.

4.6.2 Static Footprinting

In this section, we evaluate the static footprinting technique presented in Section 4.5. As explained earlier, static footprinting estimates the footprint of an operation. In contrast, dynamic footprinting reveal its actual footprint. The be of any use, footprint should not always cover the whole model (Section 4.6.2). Furthermore, static footprinting must produce estimate that are (a) valid (Section 4.6.2), (b) precise (Section 4.6.2) and (c) efficient to compute (Section 4.6.2). These results were presented in [JGB11a], but they are included (almost verbatim) in this article to make it self-contained.

Model Usage

Question 1. *What is the average model usage of a set of operations?*

The model footprint of an operation is the set of elements touched by this operation. If we have a set of operations, we define the footprint of this set as the union of the footprints of each individual operation. We can measure the model usage of an operation with the *usage ratio* η, which has two components: η_o is based on the number of objects, while η_s is counting the number of settings.

$$\eta_o = \frac{\text{\# objects in footprint}}{\text{\# objects in model}}$$
$$\eta_s = \frac{\text{\# settings in footprint}}{\text{\# settings in model}}$$

The higher these values, the more a model operation uses the elements in the input model. In the extremes, $\eta = 1$ means that the operation uses the model completely, while $\eta = 0$ indicates an empty footprint.

In Section 4.3, we present a Petri net and a query to be executed. The Petri net contains 10 objects and 40 settings. During the execution of the query, the operation has touched 10 objects and 21 settings (see Section 4.5). Thus, the model usage of the query is $\eta_o = 1$ and $\eta_s = 0.52$.

Table 4.3 presents the average (median) model usage while Figure 4.14 displays the usage ratio measured in our sample of models with respect

Table 4.3: Average model usage per operation.

	E2KV	E2SQL	E2HTML	E2DOT	E2GEN	E2* Suite
η_o	16.58%	49.43%	55.56%	51.70%	51.70%	55.86%
η_s	4.46%	25.43%	21.47%	22.57%	8.38%	39.25%

to the operations (based on dynamic footprints). The data set includes 450 measures (6 operations, 75 models). Every point represents the usage of one model by one operation. On the horizontal axis, we measure the usage in terms of objects (η_o), while the usage in terms of settings (η_s) is measured on the vertical axis. The plot is to be interpreted as follows: the higher / more right a point, the more an operation uses the elements of a model (in terms of settings respectively in terms of objects).

For example, when measuring the representation of UML in Ecore (Figure 4.15), we found that only 3% of its objects and 1% of its settings are used when creating a visitor in Kermeta for UML (E2KV). Note that many of the unused elements are used by other operations. Indeed, with respect to E2HTML, the model usage of UML increases to 67% in terms of objects and to 26% in terms of settings.

In average (median), 56% of the objects and 36% of the settings in a model are used with respect to our operation suite E2* (see Table 4.3). The low usage of E2KV can be explained by the fact that this operation is only interested in `EClass` objects and the inheritance relationships among them but not their content (such as `EAttribute` or `EOperation` objects). On the opposite, E2HTML considers almost

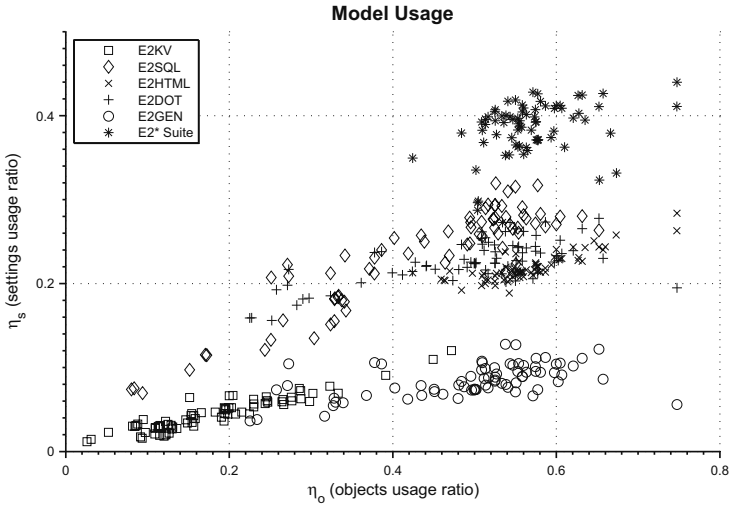


Figure 4.14: Model usage of 75 models by 6 operations.

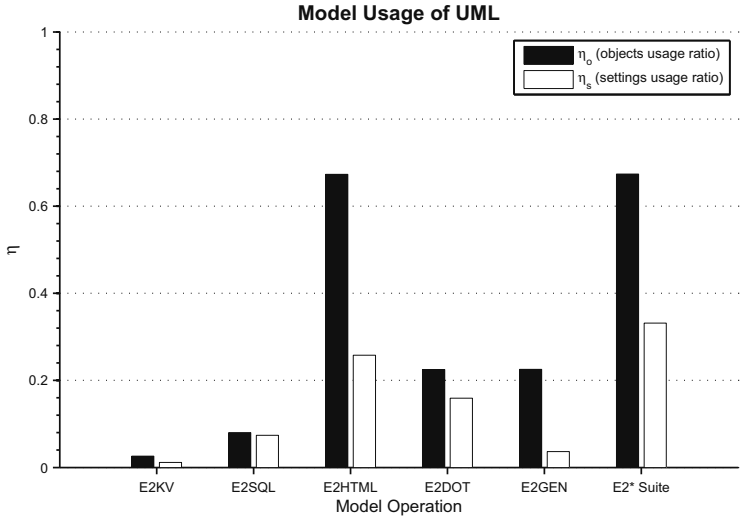


Figure 4.15: Usage of UML.ecore by 6 operations.

every kind of model element (including some of their annotations). Nevertheless, we found no footprint that completely covers a model, because none of our operation uses `EGenericType` objects.

Validity of Static Footprints

Question 2. *Does static footprinting produce valid estimate of footprints?*

To be of any use, static footprints must be conservative estimates of dynamic footprints. This property should hold by construction, but we nevertheless verified that our implementation satisfies this requirement by testing it with 450 test cases (1 test case per footprint). We executed the operations on both static footprints and complete models and compared the outcome of these executions. We found no difference between the outputs of these executions. In other words, all elements needed by the operations were indeed included in the static footprints. Note that E2GEN (and, consequently, E2*) created slightly different outputs, because the operation creates decorator models containing references to the input models. Thus, when E2GEN is executed on a static footprint, the output model refers to the static footprint rather than the complete model. This difference is insignificant and does not argue against the validity of static footprints.

Precision of Static Footprints

Question 3. *How precise is the estimation made by static footprinting?*

Since static footprints estimate dynamic footprints, we can assess the pertinence of this estimation by using the precision measure from the information retrieval field. For this purpose, we consider elements of the dynamic footprint as *relevant* while elements from the static footprint form the set of *retrieved* elements. *Precision* measures the proportion of retrieved elements that are indeed relevant. Note that the dual of precision, *recall* (the proportion of relevant statements that have been retrieved), is always trivial in this context (100%), because a static footprint is always a superset of the dynamic footprint it estimates. The precision of the estimation is measured by considering objects (σ_o) and settings (σ_s). The larger the measure σ, the closer is the static footprint to the dynamic footprint and the more accurate is the measure of model usage when it is based on static footprinting ($\widehat{\eta}$).

$$\sigma_o = \frac{\text{\# objects in dynamic footprint}}{\text{\# objects in static footprint}}$$
$$\sigma_s = \frac{\text{\# settings in dynamic footprint}}{\text{\# settings in static footprint}}$$

In our explanatory example (see Section 4.3), both footprints contain 10 objects. In terms of settings, the dynamic footprint has 21 settings

Table 4.4: Average precision of static footprints.

	E2KV	E2SQL	E2HTML	E2DOT	E2GEN	E2* Suite
σ_o	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
σ_s	89.26%	92.48%	92.89%	95.80%	65.71%	94.12%

while the static footprint contains 26 settings. Therefore, the precision of the static footprint is $\sigma_o = 1$ and $\sigma_s = 0.81$.

Figure 4.16 presents the precision of our 450 static footprints in comparison with their dynamic counterparts. On the horizontal axis, we measure the precision in terms of objects (σ_o), while the precision in terms of settings (σ_s) is measured on the vertical axis. Therefore, the higher / more right a point, the closer is the static footprint to the dynamic one.

In average, static footprints are very precise: the majority of static footprints contain no irrelevant objects (the median of σ_o is 100%) while still containing some irrelevant settings (see Table 4.4).

On the left part of the plot in Figure 4.16, there are 10 static footprints with a precision $\sigma_o < 0.8$. These are the footprints of five models (friends, BPMN, filesystem, flowchart and fsmStatic) with respect to two operations (E2HTML and E2*). The imprecision of these static footprints is due to `EAnnotation` objects. E2HTML (and consequently E2*) reads annotations whose *source* is "genmodel". These five models have a lot of annotations, but these annotations have different sources (they are destinated to other operations or

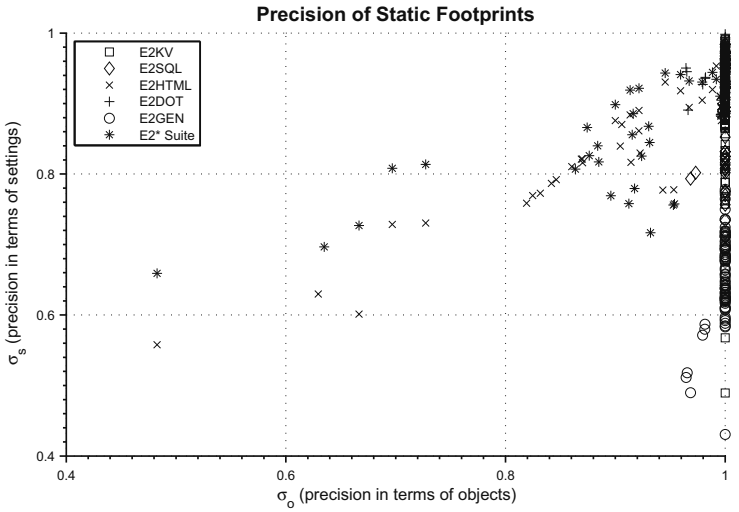


Figure 4.16: Precision of static footprints with respect to dynamic footprints.

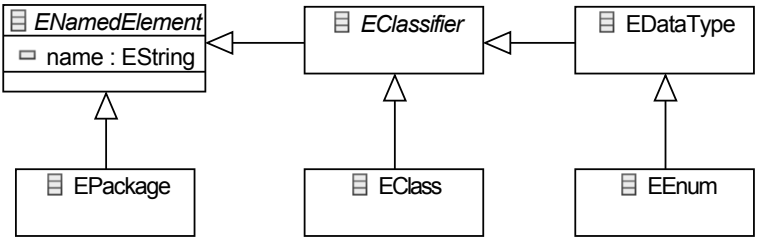


Figure 4.17: Excerpt of the Ecore metamodel: ENamedElement and some of its subtypes.

relevant to other purposes). Dynamic footprints do not include the entries of these `EAnnotation` objects while static footprints include them. These outliers reveal a limitation of static footprinting: its precision can be rather low for operations whose usage depends on conditions defined in terms of object states.

Many static footprints suffer from a lack of precision in terms of settings $\sigma_s < 0.6$ (lower left part of Figure 4.16). This lack of precision is due to the inheritance relationships among the types of the Ecore metamodel. An excerpt of this metametamodel is depicted in Figure 4.17: The feature `name` is defined in a class `ENamedElement`, which is the supertype of many other classes.

Two static footprints of E2KV are impacted by this problem. The metamodel footprint of E2KV contains all classes depicted in Figure 4.17. Still, the operation only reads the `name` of `EClass` objects, but not the `name` of `EDataType` objects. Since Ecore and BPMN

contain a lot of `EDataType` objects, their static footprints get imprecise in terms of settings, because these static footprints include settings for the name of `EDataType` objects, while the dynamic footprints do not include them.

E2GEN further illustrates this problem. Its metamodel footprint also contains all classes of Figure 4.17. However, E2GEN only reads the `name` of `EPackage` objects. Thus, many objects in the static model footprints of E2GEN will have a setting for `name`, while only `EPackage` objects will have a setting for it in the dynamic footprint. This explains the low average of σ_s for E2GEN in Table 4.4.

Other operations read the `name` of each `ENamedElement` object in their static footprints. Thus, their static footprints are not impacted by this issue.

Efficiency of Static Footprinting

Question 4. *How efficient is static footprinting?*

In this subsection, we evaluate the cost of static footprinting and compare it to the cost of dynamic footprinting. Table 4.5 summarizes the results of this evaluation. We selected four models from our sample, each one representing an order of magnitude in terms of model size: the smallest ($\sim$ 10 objects), the largest ($\sim$ 10′000 objects) and two in-between ($\sim$ 100 and $\sim$ 1′000 objects). For dynamic footprinting,

Table 4.5: Computation time of footprints.

Model	# Objects	Method	E2KV	E2SQL	E2HTML	E2DOT	E2GEN	E2* Suite
DocBook	23	Dynamic	1'955 ms	1'587 ms	1'525 ms	1'615 ms	1'541 ms	3'193 ms
		Static	8 ms	11 ms	10 ms	10 ms	10 ms	12 ms
		SpeedUp	*244x*	*144x*	*153x*	*162x*	*154x*	*266x*
XQuery	100	Dynamic	1'832 ms	1'782 ms	1'792 ms	2'171 ms	1'637 ms	3'863 ms
		Static	16 ms	24 ms	23 ms	24 ms	22 ms	22 ms
		SpeedUp	*115x*	*74x*	*78x*	*90x*	*74x*	*176x*
XHTML	1'035	Dynamic	3'211 ms	6'460 ms	8'762 ms	4'262 ms	2'238 ms	20'495 ms
		Static	101 ms	132 ms	108 ms	90 ms	66 ms	59 ms
		SpeedUp	*32x*	*49x*	*81x*	*47x*	*34x*	*347x*
OCLUML	13'849	Dynamic	6'348 ms	12'237 ms	41'267 ms	24'417 ms	5'426 ms	90'915 ms
		Static	222 ms	344 ms	651 ms	373 ms	111 ms	240 ms
		SpeedUp	*29x*	*36x*	*63x*	*65x*	*49x*	*379x*
Static Metamodel Footprint			6'242 ms	6'733 ms	6'878 ms	6'860 ms	6'910 ms	15'524 ms

we measure the time needed for both executing the operation while keeping a trace of its execution and extracting the dynamic footprint out of this trace. On the opposite, static footprinting has an initial cost for analyzing the operation definition to extract its metamodel footprint. Once this metamodel footprint has been computed, it can be used to filter any model. Thus, we keep the cost of precomputing the metamodel footprint separated from the cost of filtering models.

When the metamodel footprint is provided, static footprint (filtering a model) is always cheaper than computing dynamic footprints, even for our smallest model and our simplest operation. The speed up ranges from 29× to 379×. When the metamodel footprint is not precomputed, static footprinting (precomputing the metamodel footprint and filtering a model) is faster than dynamic footprinting in 6 cases highlighted with shaded cells.

Execution times were measured as follows. We used the implementation presented in [JGB11b], not the slicers generated by Kompren

(see Section 4.3). Each footprint (whether dynamic, static or metamodel footprint) was computed 5 times, each time in a freshly started Java virtual machine. Table 4.5 displays the medians of these experiments. Kermeta code — model operations and the static analysis of model operations — was interpreted (and not compiled to Java). To minimize the influence of Eclipse internal mechanisms on our measure, we run Eclipse in headless mode and we forced the loading of required plugins by computing a dummy footprint before the one we were interested in. The computer used for this evaluation is an Intel(R) Core(TM) i7 @ 2.8 GHz with 8 Gb RAM running Windows 7 Professional, Java(TM) 1.6.0_20.b02 (64 bits) and Eclipse 3.5.2 with Kermeta 1.3.2.

4.6.3 Operation Validation

The metamodel footprint of an operation can be used to validate the implementation of an operation. For this, the metamodel footprint of the operation must first be specified. Then, the metamodel footprint is derived from the implementation. Finally, the specified metamodel footprint is compared with the actual metamodel footprint. Deltas (differences between the actual and the specified metamodel footprint) can be classified into four categories as explained in Section 4.2.4:

Specification Error (SE): The specified metamodel footprint lacked a construct that is indeed needed for the operation.

Implementation Error (IE): The implementation of an operation does not use a construct it should (or is using a construct it should not).

Alternative Implementation (AI): The implementation uses a different set of constructs than specified, yet it delivers an equivalent result.

Possible Improvement (PI): The implementation does not use a specified construct, yet delivers the correct result according to the functional specification.

We conducted some experiments to evaluate whether this approach (a) is feasible (Section 4.6.3), (b) can detect implementation errors (Section 4.6.3) and (c) does not require too much effort (Section 4.6.3).

In these experiments, we considered two operations: E2DOT and E2SQL. For each operation, we prepared a functional specification and the metamodel footprint of a faulty implementation. For the functional specification, we prepared a valid pair of input and output as example. For this, we devised a small model representing the domain of a bank (see Figure 4.18) which was designed to cover as many Ecore (the input metamodel) constructs as possible. The faulty implementations were derived from correct ones by (i) altering a rule to use the wrong attribute, (ii) removing a rule concerning an attribute, (iii) removing a rule concerning a reference and (iv) removing a rule concerning a class. These bugs introduced 6 differences in the metamodel footprints (Table 4.6).

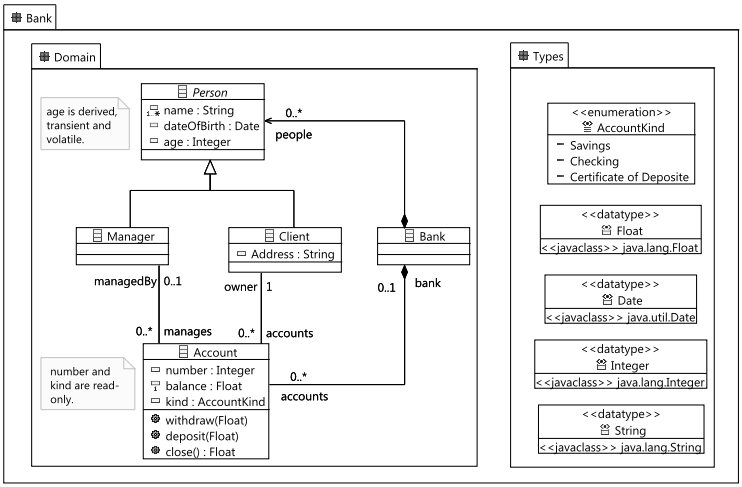


Figure 4.18: Ecore model used as input in the specification of operations.

Table 4.6: Deltas related to bugs.

Kind	E2DOT	E2SQL
wrong attribute (+)	EClass::interface	EStructuralFeature::changeable
wrong attribute (-)	EClass::abstract	EStructuralFeature::transient
attribute missing	EReference::containment	ETypedElement::lowerbound
reference missing	EPackage::eSubpackages	EClass::eSupertypes
class missing (ref)	EOperation::eParameters	EEnum::eLiterals
class missing (cl)	EParameter	EEnumLiteral

Five experts in MDE took part in this experiment. Only four of them had a strong enough experience with Ecore to complete the specification task. Thus, we only have four valid data sets. Among them, there were two doctoral students, one assistant professor and one engineer. All participants did the experiment twice, once for each operation. To keep ordering and learning effects under control, half of the participants began with E2DOT, and then proceeded with E2SQL, while the other half began with E2SQL and then proceeded with E2DOT.

First, we asked the participants to specify the expected metamodel footprint of the operation. For this, we asked them to mark, on a diagram of the Ecore metamodel, which constructs (classes, attributes, references and operations) they thought are needed by the operation. Then, we provided the participants with the metamodel footprint of a faulty implementation. We asked them to compare this metamodel footprint with the one they had just specified and discuss the identified deltas to classify them. Finally, we interviewed the participants once they had completed the experiments.

Feasibility

Question 5. *Can the metamodel footprint of an operation be specified before it is implemented?*

The key difficulty in our approach is to specify the metamodel footprint of the operation. It indeed requires good knowledge of the metamodel and the operation. In the experiments, one person did not know Ecore well enough to perform the task. When asked about the difficulties they encountered, the other participants mentioned the following uncertainties:

- implementation of loops: should the operation iterate on the references, or should it use the provided operations?
- the properties of `EStructuralFeature`
- the use of derived references
- the use of container references (`EObject` has an equivalent `eContainer()` operation).

These uncertainties are likely to show up as AI-deltas in the subsequent comparison. Sometimes, the specified metamodel footprint was not consistent. A metaclass was selected, but not the metareference needed to navigate to instances of that metaclass, or vice-versa. These inconsistencies result in deltas classified as SE.

These uncertainties and errors can be addressed with an iterative process alternating specification and implementation. At each iteration, deltas classified as SE, AI and PI can be addressed by changing the specified metamodel footprints, while deltas sorted as AI, IE and PI can be addressed by improving the implementation of the operation. A participant remarked that the hints provided by our approach were

similar to the ones a tool like check-style could provide: they may indicate real issues, but if they do not, they can be turned off.

In the interview after the experiment, all participants agreed that the approach is both useful and easy to use. They will use it when writing model operations, but only if the effort required by the approach is further reduced by automating the comparison (see below). Interestingly, most participants agreed that they internally use a similar approach mentally. However, making the metamodel footprint explicit improves the reliability of the comparison. It also has the advantage to enable the discussion among stakeholders about which types and features are to be used and which ones are to be ignored.

Detection of Bugs

Question 6. *How many deltas are related to implementation errors?*

To be of any use, the approach must be able to report the presence of bugs in the implementation without too much noise. In other words, the comparison between the specified and actual metamodel footprint must reveal (a) as many IE-related deltas as possible and (b) as few non IE-related deltas as possible. Table 4.7 displays the average number of deltas identified, classified in the four categories. In overall, participants achieve a precision of 30% and a recall of 73%.

Table 4.7: Average number of deltas.

	1st Op	2nd Op	E2DOT	E2SQL	Overall
SE	4.5	0.75	2.5	2.75	5.25
IE	4.25	4.5	4.25	4.5	8.75
AI	7.25	5.5	6.5	6.25	12.75
PI	2.25	0	0.25	2	2.25
Total	*18.25*	*10.75*	*13.5*	*15.5*	*29*
Precision	*23.29%*	*41.86%*	*31.48%*	*29.03%*	*30.17%*
Recall	*70.83%*	*75.00%*	*70.83%*	*75.00%*	*72.92%*

Interestingly, the number of deltas not related to implementation errors decreased between the first and second operation. The average number of implementation errors detected also increases for the second operation. This improves the precision from 23% (for the first operation) to 42% (for the second operation). This may be imputed to a learning effect. Indeed, both operations are implemented in a similar way. Thus, when the participants have seen the metamodel footprint of the first operation, they are less likely to commit specification mistakes, to propose alternative implementations or to suggest possible improvements.

Each operation had six IE-related deltas. The maximum number of IE-related delta discovered by a participant is five, the minimum is two. For both operations, all six deltas were uncovered by at least one participant (see Table 4.8). Interestingly, some participants (such as P2) were parsimonious when specifying the metamodel footprint, while others were quite generous (*e.g.*, P4). The former are good at

Table 4.8: Detection of IE-related deltas by the participants.

Kind	E2DOT	E2SQL	P1	P2	P3	P4
wrong attribute (+)	2	3	1	2	2	0
wrong attribute (-)	3	1	1	0	1	2
attribute missing	3	4	2	1	2	2
reference missing	2	3	2	1	0	2
class missing (ref)	3	3	2	0	2	2
class missing (cl)	4	3	2	1	2	2

identifying superfluous constructs, while the latter are good at detecting missing constructs in the actual metamodel footprint.

Effort Required

Question 7. *How long does it take to specify a metamodel footprint? How long does it take to compare the actual with the specified metamodel footprint?*

In average, participants took 9.5 minutes to specify the metamodel footprint of the first operation and 8.5 minutes to specify to metamodel footprint of the second operation. When grouped by the operation, the averages are 9.25 for E2SQL and 8.75 for E2DOT. For the comparison phase, participants took in average 19.75 minutes for the first operation, while they only needed 9 minutes for the second operation. The main reason for this difference is that participants were explained how to classify deltas during the comparison of the first operation.

The approach can benefit from tool-support in both the specification and comparison phase. In the specification phase, the (meta)modeling tool could automatically select a metaclass when a metareference pointing to it is selected. This would reduce the effort needed to specify a metamodel footprint and ensure that the metamodel footprint is consistent, reducing the number of SE-related deltas. The comparison phase can be automated with model comparison tools, such as the EMF Compare plugin[5]. For this, the metamodel must be pruned according to these metamodel footprints as explained in Section 4.4. Then the comparison of these pruned metamodels will reveal the differences between the metamodel footprints. Automating the comparison reduces the effort required by it and improves its reliability.

4.6.4 Threats to Validity

We have only evaluated our approach with operations written by us. Thus, the evaluation of our approach may not be reliable and results may differ with a different set of operations. Our experiments require both models and operations. If we had used published model operations (such as those presented in [CM00] or [KAER06]), we would have had to create UML models. Creating models rather than model operations would have been an equally strong threat to the validity of our experiments. Still, to mitigate this threat, we

[5] `http://www.eclipse.org/emf/compare/`

either based our operations on existing tools (E2DOT, E2GEN and E2HTML) or benchmarks from the MDE community [BHRV08] (E2SQL) or used their results for implementing our approach (E2KV). Furthermore, the source code of these operations can be found in [JGB11b].

We only considered Ecore metamodels and operations written in Kermeta, because of the availability of many metamodels written in Ecore. There is no apparent reason to believe that static footprinting would be less precise or less efficient in other settings (*e.g.*, models of software systems expressed in UML). Similarly, the difficulties encountered to specify metamodel footprints are essentially the same, no matter which metamodel serves as basis for the operation. Participants were not exposed to the source code of the operation; the results would therefore have been similar if we had use operation written in another language. Yet, the limited scope of our experiments remains nevertheless a threat to its external validity.

Only a few participants participated in the experiments about the validation of operation, which threatens the validity of our conclusions. We chose to invite few MDE experts instead of involving a large group of students because we wanted to collect qualitative data about the usefulness of our approach and gather feedback for future work.

Table 4.9: Summary of the results.

Scenario	Research Question	Results
Model validation	Q1: Model usage	56% of the objects and 26% of the settings used by E2*
	Q2: Validity of static footprints	Yes
	Q3: Precision of static footprints	100% in terms of objects and 94% in terms of settings for E2*
	Q4: Efficiency of static footprints	Speedup between 29x and 379x
Operation validation	Q5: Feasibility	Yes, but requires knowledge about the metamodel
	Q6: Detection of bugs	30% precision and 73% recall
	Q7: Effort required	9.5 minutes for the specification, 9 minutes for the comparison

4.7 Discussion

The results of our evaluation are summarized in Table 4.9. Our evaluation is built in two parts. One part focuses on the use of metamodel footprints for validating models. This part consists of experiments involving real-world models, but no software engineers. Its results are mostly quantitative. In contrast, the second part focuses on the use of metamodel footprints for validating operations. It involves some software engineers and its results are rather qualitative. Both parts demonstrate the usefulness and the effectiveness of metamodel footprints for their respective scenarios.

A further evaluation would assess quantitatively the extent to which footprinting helps modelers to create better models than without it. In fact, we conducted such an evaluation and reported its results in [JGBC12]. The study is a pair of controlled experiments involving students from Rennes and Zurich. Participants were asked to perform two modeling assignments. For one of them, they were given the metamodel footprint. In these experiments, we failed to demonstrate

any significant benefits of footprinting on model quality because we made some mistakes when designing the experiment: we did not train the participants, we did not provide them with proper tool support and the assignments we used in the experiments were too simple. Thus, the results of the study are inconclusive: they do not support footprinting, but they do not reject it either.

Static footprinting estimates a dynamic footprint by considering types only. Therefore, static footprints become imprecise when an operation definition involves classes that have many subclasses. In addition, static footprinting ignores conditions expressed in terms of objects states. For example, the footprint of an operation working only on an `EPackage` called "persistence" will produce a static footprint containing all packages. Improving the precision of static footprints by using a more sophisticated static analysis and filters than presented in this paper is left for future work.

In a similar direction, if a model operation relies on a reflection mechanism (*e.g.*, an interpreter which, given an OCL constraint and an UML model, evaluates the constraint on the UML model), the static footprint will be the complete model, as it is impossible to infer, from the model operation definition only, which objects or settings will be touched during the execution of the operation. In this case, one must resort to dynamic footprinting.

So far, we have implemented our approach only for operations written in Kermeta, because its imperative and object-oriented nature

makes it more accessible to industrial practices. We could implement footprinting for other transformation languages such as QVT [OMG11b] or VIATRA [VVP02]. Actually, computing the metamodel footprint of an operation written as a graph transformation is almost trivial: it suffices to collect metamodel constructs present in the left-hand side argument of each transformation rule.

4.8 Related Work

Many frameworks have been proposed to define and evaluate the quality of models. Among others, a good model is at the right level of detail [DOJ+93] for its purpose. For Lindland [LSS94], a model is semantically correct if it contains all statements that are correct and relevant for the problem at hand (completeness), but nothing more (validity). In [SR98], Schuette and Rotthowe propose the minimalism criteria to operationalize their principle of construct adequacy. A model is *minimal* if none of its elements can be removed without a loss of information for the potential model users. These criteria rely on the evaluation of the relevance of the content of the model to the problem at hand, but, to the best of our knowledge, no objective measures has yet been proposed to quantify this attribute. As Davis *et al.* point out [DOJ+93], this attribute is difficult to measure because it is highly scenario-dependent. Since more and more of activities involving models become automated (especially in a MDE environment [MA07]), we propose to use model operations

for characterizing the purpose of a model. Thus, footprints can be used to define and measure the relevance of model elements based on their usage by model operations.

Along this line, footprints are best used with editors designed to visualize a single underlying model from various viewpoints, such as the Adora editor [GBJ02] or the orthographic modeling environment [AS08]. Their visualization techniques can hide model elements irrelevant for a given model operation, producing a view specifically tailored for it.

A metamodel footprint documents the usage of an operation. Many modeling methods prescribe which kind of details is to be modeled for a given perspective (*e.g.*, [Kru95] or [RW05]). Sometimes, the creators of operations document explicitly which elements their operation uses (*e.g.*, Section 3 of [CM00]). Static footprinting not only generates automatically this documentation from the definition of operations, it can also be used to identify model elements that are excessive according to this documentation or suggest missing elements.

Furthermore, model footprints can be used for impact analysis, *i.e.*, deciding whether a change in a model impacts the outcome of the operation. In [Egy06] for example, Egyed uses dynamic footprints of consistency rule instances (these footprints are called *scope* in [Egy06]) to decide whether a given rule instance (that is, a rule evaluated with respect to a given model element) must be reevaluated after a change

in the model. Later, he extended his technique to generate fixes for inconsistencies [Egy07]: Since the scope of a rule instance contains all elements that affect its truth-value, at least one of these elements must change to fix the inconsistency. With these papers, Egyed demonstrated the advantages of instance-based incremental consistency checking over type-based incremental consistency checking. In our work, we are interested in operations applied to the model in its entirety, which typically requires more time to execute than verifying some conditions on some set of elements. Thus, we are interested in estimating footprints statically, rather than tracing the execution of the operation.

Metamodel footprint can be used to validate the implementation of an operation. Küster *et al.* lists metamodel coverage as a possible fault. In our approach, the goal is not to cover the whole metamodel, but only those constructs part of the specified metamodel footprint. This motivated Wang *et al.* to propose a tool analyzing the metamodel coverage of model transformations [WKC06]. In contrast to their analysis, metamodel footprints can be computed from model operations written in imperative languages.

In [HRK11], Hermannsdoerfer *et al.* analyzes the usage of metamodel constructs by a set of models built with the metamodel. The metamodel can then be improved based on such analysis. In our work, we focus on the relationship between an operation and it supporting metamodel and the relationship between an operation and its input models.

Formally, static footprinting consists of 3 slicing operations. The metamodel footprint of an operation is a list of types and features extracted from the definition of the operation. This list can then be used to (a) extract a slice of the metamodel for visualization purpose or (b) to extract a slice of the model to estimate the footprint of the operation. These slicers can be specified with Kompren, a DSL to specify model slicers [BCBB12].

To visualize a metamodel footprint, we slice the input metamodel. Metamodel pruning [SMBJ09] produces a similar slice of metamodels, called *effective metamodels*. An effective metamodel includes more constructs than those strictly required to visualize a metamodel footprint. These constructs (such as mandatory properties or opposite references) are needed to make sure that the effective metamodel is a supertype of the complete metamodel. In our work, we present how metamodel footprints can be used to estimate model footprints or to validate model operations. In [SMM+12], Sen *et al.* present another use for metamodel footprints and effective metamodels: the reuse of operation across metamodels. Once the metamodel footprint has been identified, it becomes possible to map the constructs it contains to constructs from one metamodel to another metamodel.

Static footprints are slices of models. Unlike other model slices (such as the one presented in [KSTV03]), our slicing criterion is not defined in terms of the behavior depicted by the model, but is related to the behavior of the model operation being executed on it.

4.9 Conclusion and Future Work

A model operation typically only touches some part of the model during its execution, which forms the footprint of that operation. Establishing the footprint of some operations with respect to a model supports modelers in reducing the scope and decreasing the level of details in the model so that it only contains the information that impacts the outcome of these operations. Moreover, footprinting gives hint for finding defects in operations and may suggest improvements for them. Similarly, an operation usually only uses a fraction of the metamodel for which it is defined. The part of a metamodel involved in its definition forms its metamodel footprint. The metamodel footprint can be used to validate and complete models and operations in a MDE context.

The footprint of an operation can be revealed with dynamic footprinting, requiring its execution. However, footprints can be estimated with static footprinting, using the metamodel footprint of the operation as a filter. We evaluated static footprinting in an experiment involving 75 models and 5 operations (+1 combining these operations). This experiment suggests that static footprinting can estimate dynamic (actual) footprints with a high precision (in average, 100% in terms of objects and 94% in terms of settings for E2*). Furthermore, static footprinting is between 29 and 379 times faster than dynamic footprinting when the metamodel footprint of an operation is precomputed.

Metamodel footprints can also specify model operations, providing a starting point for their implementation and enabling their validation. In an experiment, MDE experts were able to detect most bugs we introduce in operations with the help of metamodel footprints, without having to look at the source code of the operations and without having to execute them.

We believe that footprints increase the confidence of modelers that they are modeling the right thing and the confidence of engineers that their operation is using the right thing. However, our work is only an initial step in analyzing which parts of a model or a metamodel are used by model operations. The precision of static footprints can be improved as future work. Since this analysis is likely to be more sophisticated than the one presented here, tool support for footprinting will be even more important, suggesting another direction for future work.

Chapter 5

Impact of Footprinting on Model Quality

Original publication:
Impact of Footprinting on Model Quality: An Experimental Evaluation
C. Jeanneret, M. Glinz, B. Baudry and B. Combemale
Model-Driven Requirements Engineering Workshop at RE 2012

Abstract

When modeling requirements, software analysts have to choose the relevant modeling constructs among all those available. If they do not choose the right set, their model may lack some important information or their model may contain many superfluous details. In previous work, we proposed to capture the purpose of a model with a set of model operations such

as queries or model transformations. Then, modelers can analyze the footprints of these operations, that is, the set of model elements touched during their execution.

In this paper, we report on two controlled experiments performed with students to evaluate whether footprinting can help them in creating better models. While our studies did not demonstrate statistically significant benefits of footprinting, they reveal the importance of training and tool support for the analysis of footprints.

5.1 Introduction

With the advent of Model Driven Engineering (MDE), models play an important role in the development of software. Conceptually, a model is an abstract representation of an original for a given purpose. Ideally, a model exactly represents its *domain*, that is, the set of all possible statements that would be correct about the original and relevant for the purpose at hand [LSS94].

A model can differ from its domain in two different ways. In [LSS94], Lindland et al. define validity as the extent to which a model only contains statements from the domain. A model with superfluous details may be harder to understand and is probably more expensive to create than necessary. On the opposite, a model may lack some relevant details, making it incomplete. When using an incomplete model, an interpreter may draw wrong conclusions about the original.

Thus, the value of a model depends heavily on whether it contains all relevant information and only this information. However, creating models that actually represent their domain is not an easy task. Eliciting information about the original is not enough; the modeler needs to understand the purpose of the model as well. Often, little is known about this purpose and typical modeling assignments only mention the modeling language to be used. When the language contains many constructs, as UML does, this indication offers little help.

Recently, we have invented a technique called footprinting to detect the presence of superfluous elements in models [JGB11a]. In a MDE setting, the purpose of a model can be characterized by the set of model operations (*e.g.*, queries, view extractions or model transformations) that the model must enable. A *footprint* is the set of all model elements that have been used during the execution of these operations. When footprints are highlighted, modelers can easily find the elements that were not used by the set of operations executed on the model, which gives hints about their possible irrelevance. However, the benefits of footprinting have not yet been evaluated empirically.

In this paper, we are interested in the actual effect of footprinting on the quality of models. To investigate this effect, we performed a pair of controlled experiments: one at the University of Zurich, Switzerland, involving undergraduate students enrolled in a basic Software Engineering course, and one at the University of Rennes 1, France, involving graduate students enrolled in a course on MDE techniques.

The remainder of the paper is structured as follows. In the next section, we provide background information about model quality and model footprinting. We describe our experiments in Section 5.3, while their results are presented and analyzed in Section 5.4. Finally, in Section 5.5, we discuss our findings and we conclude in Section 5.6.

5.2 Background

5.2.1 Model Quality

As models become more and more important for the development of software, many researchers proposed frameworks for evaluating and improving the quality of models. In [LSS94], Lindland et al. proposed the first systematic approach to identifying quality goals and means to achieve them. In this framework based on semiotics (the study of symbols), models are seen as sets of statements. Models are compared to three other sets of statements: The *language* is the set of statements expressible in the modeling language. The *domain* is the set of all possible statements that would be correct and relevant while the *audience interpretation* is the set of statements that the model users think a model contains.

Their framework defines three types of model quality. *Syntactic quality* is how well a model corresponds to the language. *Semantic quality* is how well the model corresponds to the domain. Finally, *pragmatic quality* is how well a model corresponds to the audience interpretation. We chose the framework of Lindland et al. for our work because it laid the foundation for many other publications. Krogstie *et al.* extended this framework with additional constructs from the semiotics theory [KSJ06]. Moody *et al.* validated the framework empirically in [MSBS03] while España *et al.* used the framework to compare two RE methods in [ECFGP09].

Lindland *et al.* distinguish two semantic quality goals: completeness and validity. A model is *complete* if it contains all the statements from the domain, while a model is *valid* if it only contains statements from the domain (and nothing else). A statement can be invalid for two reasons: it is incorrect or it is irrelevant for the purpose at hand. In our work, we distinguish between the two cases. We use the term *confinement* for the extent to which a model only contains relevant statements [MDN09] while *correctness* designates the extent to which a model only contains correct information.

In typical settings, the domain does not exist explicitly. Semantic quality is then typically assessed subjectively with reviews. In our experiments, the domain is represented by a reference model. This has been done previously by España *et al.* to compare two RE methods in [ECFGP09]. Furthermore, we not only consider the actual quality of models — measured on the model with respect to the reference model — but also the quality as it is perceived by the modelers. This perception is important, because on a typical modeling task, the domain does not exist explicitly and there is no such thing as a reference model. Thus, modelers consider their modeling task complete when their models contain all statements from what they perceive as the domain.

5.2.2 Model Footprinting

To assess the relevance of some model elements, the modelers must know the purpose of their models. In previous work [JGB12], we

proposed to use goal oriented requirements engineering methods to capture this purpose. Eventually, this purpose must be operationalized with some model operations such as analyses, queries and model transformations. For example, a class diagram could be used to derive a glossary of a domain or a state machine could be used to generate the skeleton of an implementation. The main assumption behind the idea of footprinting is that the purpose of a model can be characterized by the set of model operations that will be performed on this model.

These operations can be performed mentally by humans or executed by a computer. In both cases, a model must conform to some structure to support these operations. This structure is typically expressed with an object-oriented metamodel. Such a metamodel defines the types and features — meta-classes, meta-attributes and meta-references — that can be used in a model. For example, the metamodel of UML defines meta-classes such as `Class` and `State`, and meta-attributes such as `name` and `isAbstract`.

Footprinting consists of finding which elements of a model are used by a given model operation during its execution. There are two techniques for footprinting: dynamic and static footprinting [JGB11a]. Dynamic footprinting analyzes the execution of a set of operations, while static footprinting estimates the footprint based on a static analysis of the definition of the operations. In [JGB11a], we demonstrated that, despite some limitations, static footprints are very precise estimates of dynamic footprints and yet cheap to compute.

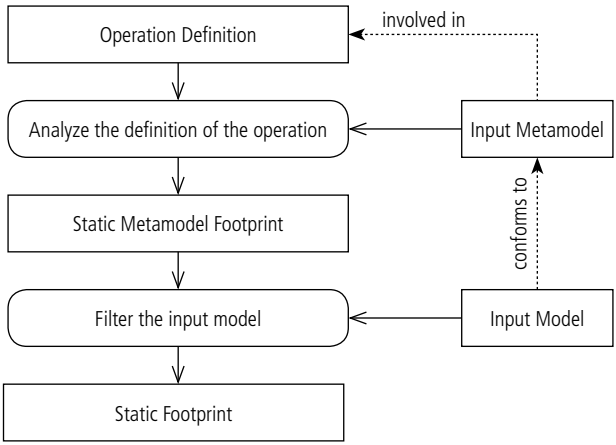


Figure 5.1: Static footprinting.

In the experiments, we have used static footprinting, as it can be done without tool support. Static footprinting is divided into two phases (see Figure 5.1). First, the source code of the operations is analyzed. Types and features are collected along their control flow graph. This analysis yields a *static metamodel footprint*, the set of all constructs relevant for the operations being analyzed. In the second phase, models are filtered through the static metamodel footprint: only elements related to constructs in the static metamodel footprint are kept in the *static (model) footprint*. This filtering is very simple and can be done manually. Thus, as long as the static metamodel footprint is provided, static footprinting can be done without tool support.

For example, the operation generating a glossary from a class diagram may be implemented as a model-to-text transformation. The source code of this operation may involve classes and generalizations, but probably not ports or dependencies. In addition, the operation will likely read the name of a class, but not whether it is abstract. The static metamodel footprint would then include the following elements:

- `Class`
  - `name`
- `Generalization`

The modelers should only model elements that are instances of the static metamodel footprint. Any other elements will not be used by the operations performed on the model. They may therefore be irrelevant, decreasing the confinement of the model. Furthermore, the modeler should consider all constructs in the static metamodel footprint. If a relevant meta-class or meta-attribute is omitted from the model, some information is probably missing, decreasing the completeness of the model. The filtering step of static footprinting validates whether a model only contains elements related to the static metamodel footprint.

We believe that with the help of footprinting, modelers produce models that are more confined and more complete than without footprinting. Indeed, static metamodel footprints make explicit, what kind of details should be modeled and what kind should not.

Besides, we expect footprinting to make modelers more confident in the completeness and confinement of their model. The purpose of these experiments is to confirm or to refute this belief. In the next section, we discuss the planning of our experiments.

5.3 Experiment Planning

The goal of our experiments is to evaluate the effect of model footprinting on the actual and perceived quality of models in the context of students modeling the requirements of a software system. Following the template defined by Wohlin [WRH+00], we present the design of the experiments we conducted towards this goal. In the next section, we present the context and the material of the experiments. Then, in Section 5.3.2, we highlight the variables of our experiments and formulate our hypotheses. In Section 5.3.3, we lay out the design of our experiments and we discuss threats to validity in Section 5.3.4.

5.3.1 Context and Material

Our experiments were carried out at two sites: Rennes and Zurich. In Zurich, the context of our experiment is a second year (undergraduate level) Software Engineering (SE) course at the University of Zurich. As a prerequisite for this SE lecture, participants had to follow a lecture

on modeling[1]. While this modeling lecture does not aim at teaching UML per se, many UML diagrams are covered during the lecture. Students learn the various constructs and modeling guidelines, and they apply this knowledge on some practical exercises. In Rennes, the experiment took place within a Model Driven Engineering (MDE) lecture, which is visited by two second year classes on the graduate level: Miage (oriented towards business) and GL (oriented towards software engineering). The students from Rennes had to visit a lecture on modeling as prerequisite too[2]. At both sites, the experiment was part of a series of compulsory labs and exercises. Students were told that they would not be graded on performance, but that they were expected to solve their modeling assignments in a professional manner to obtain the points assigned to the lab.

Alternatively, we could have recruited these students with some financial incentives. According to Sjøberg *et al.* [SAA+02], our experiment would have been easier to organize with this alternative. However, some students could have refused to participate in our experiment, reducing the number of data points and introducing a bias towards motivated students.

Participants were not aware that we were attempting to evaluate the impact of footprinting, but knew that we would evaluate the quality

[1]This lecture is taught by Prof. Martin Glinz. The description and the material of the module can be found at `http://bit.ly/P8RrNZ` (in German)

[2]This lecture is taught by Dr. Noël Plouzeau. The description of the module can be found at `http://bit.ly/OdIKBU` (in French)

of their models by comparing them with a reference model. The experiments were monitored by the first author on both sites. He was not involved in any of the courses as an assistant, reducing the bias of the experiment. During the experiment, students were not allowed to talk to each other.

The modeling assignments were solved with MagicDraw[3], an UML modeling tool. We chose MagicDraw over other modeling tools because it fully supports UML class diagrams and state machines and it is intuitive to use. Using a tool makes the modeling task more realistic than pen-and-paper only [SAA+02]. It has the additional advantage that it prevents participants from making syntactic mistakes. While the tool is used by the students from Rennes during their curriculum, students from Zurich never used it before the experiment. To overcome this issue, we answered all the questions related to MagicDraw during the experiments. Furthermore, we let students use comments to specify UML elements when they did not find out how to specify them properly with the tools. For example, some students modeled the triggers of state transitions by attaching comments to the transitions. We took these comments into account when assessing the quality of models.

For this experiment, we developed the following material: a modeling assignment with 2 exercises, the reference models and the corresponding static metamodel footprints. This material is included in

[3] `http://www.nomagic.com/products/magicdraw.html`

Appendix A. The exercises consist in creating UML models from case descriptions in plain text. The exercises were taken from [SJB08]. The description for the class diagram is the *dental clinic* case (on page 272) and the description for the state machine is the *shipment* case (on page 275). Students were asked to solve each exercise within 45 minutes, so that they had enough time to solve both exercises and fill in the questionnaires during the time allocated for the labs (2 hours). The textbook also provides some solutions, which were used to create the reference models.

To define the static metamodel footprints, we decided not to define (and later analyze) some operations, but rather derive the static metamodel footprint from the reference model directly. That is, we collected all types and features present in the reference model. We decided to do so to avoid a potential threat to validity: defining model operations ourselves might have introduced some bias in the experiments. However, this decision leaves the validation of the main assumption of footprinting — the purpose of a model can be characterized as a set of operations — out of the scope of this paper. This will be investigated in future work. The experiment remains nevertheless valid, as the origin of the static metamodel footprint is of no importance for modelers.

In total, students answered 4 questionnaires. First, they had to participate in a pre-experimental survey, introducing them to the experiment and collecting their modeling experience. After each modeling exercise, students had to fill in a questionnaire to report how they

perceived the modeling task, the case description and the quality of their model. They also had to mention how long they took to perform the modeling task. The post-experimental questionnaire explained briefly the goals of the experiment and invited students to provide us with feedback about the experiments.

5.3.2 Variables and Hypotheses

In our experiments, we investigate the impact of footprinting on model quality. We measured 4 quality aspects characterizing the semantic quality of a model as defined by Lindland in [LSS94]: completeness (COM), confinement (CON), correctness (COR) and overall quality (OAQ). Each quality aspect comes in two flavors: actual and perceived quality. Actual quality is the semantic quality of a model with respect to a reference model. We measure this quality by comparing the model to a reference model. Perceived quality is the quality of the model as perceived by its modeler. We measure it with questionnaires. In total, our experiments involve 9 variables: 8 dependent variables and 1 independent variable.

Actual Quality

We quantify actual quality by counting the number of mistakes in the model with respect to a reference model. We distinguish three types of mistakes. A completeness mistake is the absence of some

element from the model. For example, in a class diagrams at the analysis stage, the absence of multiplicities or a missing attribute are completeness mistakes. Confinement mistakes are due to the presence of superfluous but correct elements in the model. For example, the presence of interfaces or getter operations is a confinement mistake in an analysis class diagram. All other mistakes are denoted as correctness mistakes. This kind of mistakes includes wrong multiplicities or redundant attributes in a class diagram. ACOM counts the number of completeness mistakes, ACON counts the number of confinement mistakes, ACOR counts the number of correctness mistakes and AOAQ is the total number of mistakes.

Perceived Quality

Perceived quality is measured with a questionnaire. More precisely, students were asked to what extent they agreed on the following statements using a 5-level Likert scale:

- My model is a good model. (POAQ)
- My model contains all relevant information. (PCOM)
- My model only contains relevant information. (PCON)
- My model only contains valid information. (PCOR)

Method

All participants used the same modeling languages (UML class diagrams and UML state machines). The purpose of the model was

stated informally in all assignments. Still, some participants received additional information for one of the two diagrams they had to create: the static metamodel footprint (FP), that is, the set of modeling constructs relevant for the purpose of the model. This is a nominal variable: either the static metamodel footprint was available or not. FP is the only independent variable of our experiment.

Hypotheses

Based on the goal of the experiment, we formulate, for each of the dependent variables, the following null-hypothesis (H_0): there is no difference between the quality of models when the modeler knows (or does not know) the static metamodel footprint (FP). The alternative hypothesis (H_1) is that footprinting has an impact (positive or negative) on the quality of models.

In this paper, we use Wilcoxon rank-sum tests to test our hypotheses. A Wilcoxon rank-sum test is a non-parametric test to compare the median of two samples. We do not use the T-test because the data does not respect all its assumptions: the counts of mistakes (actual quality) follow Poisson distributions rather than normal distributions and the Likert-scales (perceived quality) only deliver ordinal values. If the non-parametric test is not powerful enough, we fit the counts of mistakes to Poisson distributions and compare their parameter.

5.3.3 Experiment Design

This experiment only involves one factor whose effect is interesting to us: whether the static metamodel footprint is given in the task assignment or not. However, a nuisance factor may impact the quality of models: the ability of students in modeling. Therefore, we opt for a completely randomized design in which each participant uses both methods (with and without static metamodel footprint) [WRH+00]. Overall, our design is inspired by those of Briand *et al.* when they investigated the benefits of OCL in UML based development [BLYDP04].

To minimize learning effects between the modeling tasks, we considered two different objects: a class diagram and a state diagram. We keep ordering effects under control by using four groups instead of two, covering all possible permutations. These groups are presented in Table 5.1. For example, participants in group A first model the class diagram without the static metamodel footprint. They then proceed to the state diagram, this time with the static metamodel footprint.

5.3.4 Threats to Validity

External Validity

In this experiment, participants are students, half-way to get their Bachelor's or Master's degree. They may not be representative of

Table 5.1: Experiment design.

		Group A	Group B	Group C	Group D
Tasks	1st	Class Diagram With Footprint	Class Diagram Without Footprint	State Machine With Footprint	State Machine Without Footprint
	2nd	State Machine Without Footprint	State Machine With Footprint	Class Diagram Without Footprint	Class Diagram With Footprint

software analysts working in the field. However, these students know UML at this stage as much as when they will leave the university and enter their professional career. Besides, footprinting is meant to help modelers who do not know much about how their models are used, which is typically the case for novice modelers.

We intentionally kept the modeling tasks small enough so that the students could achieve them within 45 minutes. This constraint reduces the (undesired) effect of fatigue on the results. However, these assignments are not representative of modeling tasks in an industrial context because of their small size and the presence of a complete case description (participants did not have to elicit requirements). Still, if footprinting has an impact on small models, it will likely have an impact on larger models, too. Elicitation was kept out of the experiment, as it is not within the scope of our investigations.

Internal Validity

Besides the availability of the static metamodel footprint, other factors can impact the quality of models, like the ability of participants and

ordering effects (learning and fatigue). We kept ability under control by having each participant use both methods, while ordering effects were kept under control by having four groups solving two different modeling tasks in different order.

We provided footprinting in its most basic form: a static metamodel footprint in textual form. Alternatively, we could have provided an example of models containing all constructs from the static metamodel footprint or a tool displaying the model footprint. We chose the most basic form to reduce the bias towards footprinting. Furthermore, the static metamodel footprint can give some hints about completeness, while the static (model) footprint cannot.

Construct Validity

The perceived quality (PCOM, PCON, PCOR and POAQ) is measured using a questionnaire. We phrased our questions so as to avoid bias. Actual quality (ACOM, ACON, ACOR and AOQ) is measured by counting the number of mistakes made with respect to a reference model, that is, the solution of the modeling assignment. This reference model was written by the first author based on the correct model provided in [SJB08]. Still, to mitigate the risk that this domain is biased, the domain was validated by other authors. The count of mistakes is not a fully accurate measure of quality, as some mistakes are more important than others. Still, we did not assign weights to them to keep the measure as objective as possible.

Students may have acted so as to please our expectancies. This threat was reduced by (a) not telling the students the exact hypotheses behind our experiments and (b) having the experiments supervised by a researcher who is not involved in the lectures participants were enrolled to (SE for Zurich, MDE for Rennes). In the next section, we present and discuss the results obtained during the experiments.

5.4 Results Analysis and Interpretation

In total, 86 students participated to our experiments: 14 Miage and 23 GL students from Rennes and 49 SE students from Zurich. From them, we gathered 72 complete datasets including both diagrams and answers to surveys. The datasets are spread almost equally in each group (19 in group A, 15 in B, 19 in C, 19 in D). Based on the pre-experimental survey, most students consider themselves as experienced with modeling and UML in general. However, the average experience is lower with state machines than with class diagrams. Most students found that both the modeling tasks and the case descriptions were clear and that they had enough time to do the modeling. In average, participants took 39 minutes for the class diagram and 32 minutes for the state machine. Footprinting had no impact on the time used for modeling. In the remainder of this section, we first analyze the impact of footprinting on actual quality and then its impact on perceived quality.

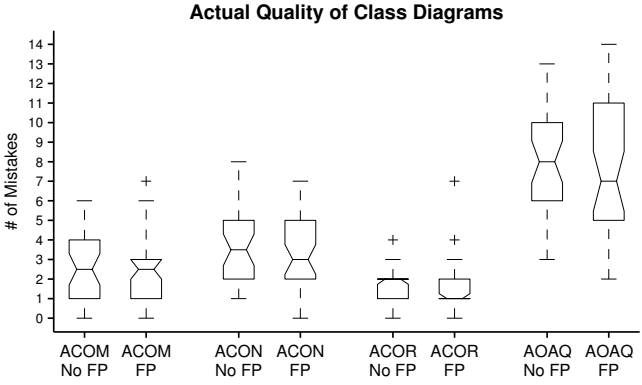
Table 5.2: Median number of mistakes.

		ACOM	ACON	ACOR	AOAQ
Class Diagram	No Footprinting	2.50	3.50	2.00	8.00
	Footprinting	2.50	3.00	1.00	7.00
	P-Value	0.91	0.41	0.20	0.63
	Reject H_0?	No	No	No	No
State Machine	No Footprinting	1.00	1.50	2.00	5.00
	Footprinting	1.00	2.00	2.00	5.00
	P-Value	0.62	0.31	0.84	0.76
	Reject H_0?	No	No	No	No

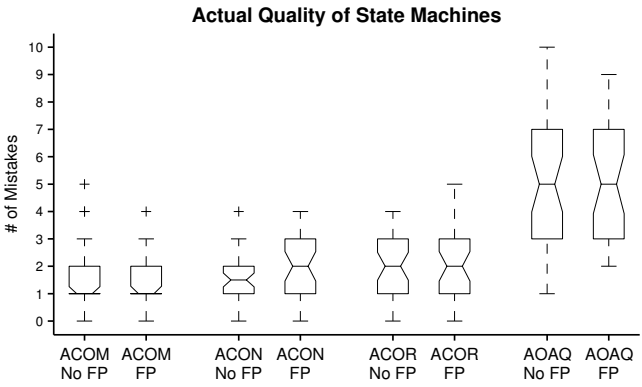
5.4.1 Actual Model Quality

To assess the actual quality of models, we compared the models to our reference models. We used more than one reference model, as the information from a textual description may be modeled correctly in different ways. Besides, we ignored differences in names, as long as the names were meaningful. Some other deviations may have been acceptable (*e.g.*, the presence of business operations in a class diagram), but we marked them all to avoid introducing some bias in the results. In the class diagrams, we identified a total of 72 possible mistakes. 31 mistakes are related to confinement, 17 to completeness and 24 to correctness. For the state machines, we have listed 52 mistakes. 16 mistakes are confinement issues, 16 are completeness problems and 20 are correctness mistakes.

The actual quality of models is presented as boxplots in Figure 5.2. Figure 5.2a shows the number of mistakes made in class diagrams,



(a) Class Diagrams



(b) State Machines

Figure 5.2: Actual quality of models.

while Figure 5.2b displays the number of mistakes made in the state machines. The results are grouped by the availability of the static metamodel footprint (the treatment) and the various quality aspects (completeness, confinement, correctness and overall quality). The y-axis represents the number of mistakes made. The higher the number of mistakes, the lower the actual quality of a model. In a boxplot, the boxes have lines at the first quartile, median, and third quartile. The notches around the medians represent their 95% confidence interval, which is useful for comparing medians.

Table 5.2 displays the median number of mistakes made. For each kind of diagrams and for each quality attribute, we compare the median number of mistakes made by students who knew the static metamodel footprint with the median number of mistakes made by students who did not. For this comparison, we use a Wilcoxon rank-sum test whose p-value is displayed in Table 5.2. We cannot reject any null-hypotheses, as no p-value is above the usual significance level $\alpha = 5\%$. Thus, we use a parametric model to further investigate whether footprinting had a statistically significant impact on the actual model quality.

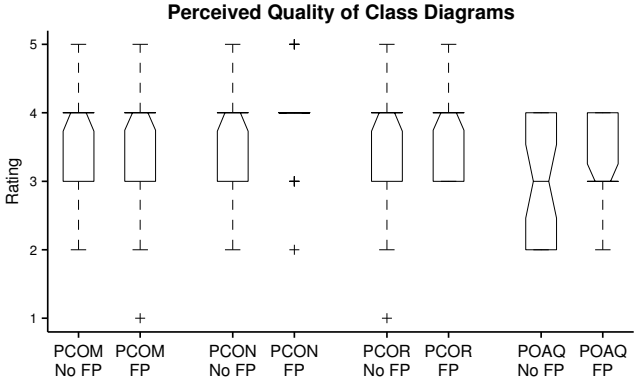
The numbers of mistakes follow Poisson probability distribution laws. Table 5.3 displays the average number of mistakes made by students in their models. The average is an unbiased estimator of the λ parameter of the Poisson law. Overall, students made 7.86 mistakes in their class diagrams and 5.13 mistakes in their state machines. Table 5.3 also provides the 95% confidence intervals for λ. The difference

Table 5.3: Confidence interval of λ for the counts of mistakes.

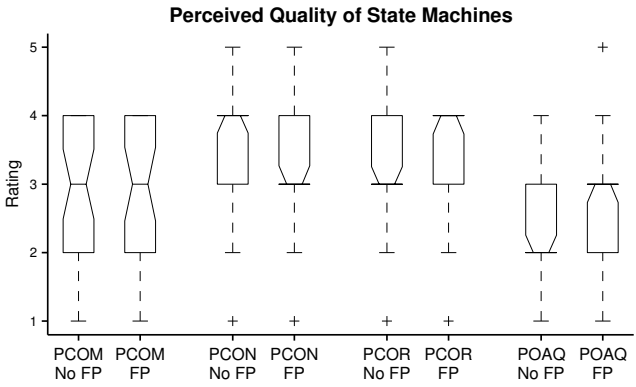
		ACOM	ACON	ACOR	AOAQ
Class Diagram	No Footprinting	2.53	3.79	1.74	8.06
		[2.02 - 3.12]	[3.14 - 4.45]	[1.32 - 2.24]	[7.10 - 9.01]
	Footprinting	2.68	3.45	1.55	7.68
		[2.16 - 3.21]	[2.86 - 4.04]	[1.18 - 2.00]	[6.80 - 8.57]
	Altogether	2.61	3.61	1.64	7.86
		[2.24 - 2.98]	[3.17 - 4.05]	[1.34 - 1.93]	[7.21 - 8.51]
State Machine	No Footprinting	1.68	1.50	1.87	5.05
		[1.30 - 2.15]	[1.14 - 1.94]	[1.46 - 2.36]	[4.34 - 5.77]
	Footprinting	1.47	1.76	1.97	5.21
		[1.09 - 1.94]	[1.35 - 2.27]	[1.53 - 2.50]	[4.44 - 5.97]
	Altogether	1.58	1.63	1.92	5.13
		[1.29 - 1.87]	[1.33 - 1.92]	[1.60 - 2.24]	[4.60 - 5.65]

between the quality with and without footprinting is not statistically significant, as all confidence intervals for λ overlap.

Interestingly, the results obtained for the class diagram contradict those obtained for the state machine. For class diagrams, footprinting improves all quality aspects except completeness. Indeed students with footprinting makes fewer mistakes in total (AOAQ) than without footprinting: fewer confinement mistakes (ACON), fewer correctness mistakes (ACOR) but more completeness mistakes (ACOM). In contrast, footprinting only improves completeness for state machines (ACOM), while degrading all other qualities (ACON, ACOR and AOAQ).



(a) Class Diagrams



(b) State Machines

Figure 5.3: Perceived quality of models.

Table 5.4: Median rating of perceived quality.

		PCOM	PCON	PCOR	POAQ
Class Diagram	No Footprinting	4.00	4.00	4.00	3.00
	Footprinting	4.00	4.00	4.00	3.00
	P-Value	0.59	0.22	0.38	0.33
	Reject H_0?	No	No	No	No
State Machine	No Footprinting	3.00	4.00	3.00	2.00
	Footprinting	3.00	3.00	4.00	3.00
	P-Value	0.73	0.28	0.57	0.23
	Reject H_0?	No	No	No	No

5.4.2 Perceived Model Quality

The ratings of perceived quality are plotted as boxplots in Figure 5.3. Given the ordinal nature of Likert scales, we use the median to characterize the perceived model quality (see Table 5.4). For class diagrams, footprinting had almost no impact on the perceived quality of models. For state machines, footprinting reduces the perceived confinement (PCON) of models, but improves the perceived correctness (PCOR) and the perceived overall quality (POAQ). However, the p-values of the Wilcoxon rank-sum tests are all above the $\alpha = 5\%$ level required to reject a null hypothesis. Thus, the differences between the medians are not statistically significant.

In general, the perceived quality is coherent with the actual quality. To assess this coherence, we group the models according to their perceived quality and we compute the average actual quality for each

Table 5.5: Coherence between perceived and actual model quality.

			Actual Quality			
			COM	CON	COR	OAQ
Perceived Quality	Class Diagram	1	5.50 (2)		1.50 (2)	
		2	3.82 (11)	2.67 (3)	2.00 (2)	9.63 (19)
		3	2.70 (20)	3.75 (16)	1.65 (20)	8.08 (25)
		4	2.18 (34)	3.62 (47)	1.68 (44)	6.46 (28)
		5	1.40 (5)	3.67 (6)	1.00 (4)	
	State Machine	1	3.40 (5)	1.67 (3)	2.25 (4)	7.00 (8)
		2	2.33 (21)	1.00 (3)	1.90 (10)	5.71 (28)
		3	1.23 (22)	1.41 (27)	1.75 (24)	4.13 (23)
		4	0.88 (24)	1.67 (33)	2.03 (32)	4.25 (12)
		5		2.67 (6)	1.50 (2)	7.00 (1)

group. Table 5.5 displays these averages. For example, 19 students assessed the overall quality of their model with a 2. The average number of mistakes for these students is 9.63. In comparison, 28 students assessed the overall quality of their class diagram with a 4 and made, in average, 6.46 mistakes. In most cases, the higher the perceived quality is, the lower the number of mistakes is, and, thus, the higher the actual quality is. We observe similar results for models made with and without static metamodel footprints. Thus, in Table 5.5, we only show the results for all models, no matter whether the static metamodel footprint was available for their construction.

5.5 Discussion

We have hypothesized that footprinting would improve the quality of models and increase the confidence of the modelers in the quality of their models. After all, footprinting provides additional hints that should have helped the participants in their modeling tasks. However, our experiments did not demonstrate any statistically significant effect of footprinting on model quality. Unfortunately, our post experimental survey did not include any question to further explain this issue. Thus, more research is needed to investigate whether this is due to the design of our experiments. In this section, we propose and discuss possible explanations, providing some direction for future research.

The modeling assignments may have been too simple to demonstrate the benefits of footprinting. Indeed, the case descriptions do not include many irrelevant details with respect to the given modeling purpose. In the post-experimental surveys, most participants disagreed that both case descriptions were too detailed. Thus, the modelers had no difficulty to figure out what was to be modeled, no matter whether or not they knew the static metamodel footprint. The results may have been different if we had used a large case description with many superfluous details. In such a case, the static metamodel footprint may help for deciding which details should be included in the model and which details should be left out.

Participants may have overlooked the static metamodel footprint or may not know how to exploit this information. Indeed, some kinds of mistakes would have been avoided if the static metamodel footprint had been used properly. The number of students committing these mistakes supports this explanation. For example, 30 students forgot to model the triggers on transitions in the state machines: 18 students had the static metamodel footprint, 12 had not. For the class diagrams, 11 students modeled realization relationships, which are superfluous for the purpose at hand: 8 of them had the static metamodel footprints, 3 had not. This also suggests that our measure for actual quality is not responsible for the results.

Future research should clarify to what extent the form of footprinting reduces its impact on model quality. We chose to provide the static metamodel footprint as text, directly integrated in the modeling task without highlighting it. Alternatively, we could have extended the modeling tool to provide the students with some feedback about the model footprint, displaying warning messages about superfluous or missing elements. We could also have presented an example of a model containing all the constructs in the static metamodel footprint. In these forms, the participants can more easily use the information provided by footprinting than in its basic form and they can less easily overlook it.

In our experiments, we did not introduce footprinting and we did not train participants to use it. Thus, participants may not be experienced enough in metamodeling or in the UML metamodel to properly

exploit the static metamodel footprint. While this may be the case for the students in Zurich, these subjects were taught to the students from Rennes in the MDE lecture. Yet, there is no significant difference in the results from both sites. Besides, most students considered themselves as experienced in modeling, in UML and in class diagrams during the pre-experimental survey. Still, footprinting may require some training before it can actually deliver some benefits to the modelers.

5.6 Conclusion and Future Work

Creating requirement models is not an easy task. In addition to eliciting information about the original, modelers need to understand the purpose of their model. Often, this purpose is kept implicit and the only indication is a modeling language. In a MDE setting, where the model is often used to feed some model operations (like queries or transformations), footprinting can be used to assess and improve the quality of models. Still, the benefits of footprinting on model quality were not yet empirically investigated.

To investigate the benefits of footprinting, we conducted a pair of controlled experiments involving students from two universities, Rennes and Zurich. Participants had to model a class diagram and a state machine. Some participants used footprinting for the class diagram, while others used it for the state machine. We evaluated both the

actual quality of models — by counting the number of mistakes with respect to a reference model — and the quality perceived by the modeler — by using a questionnaire.

Our results are inconclusive: the effect of footprinting on quality is not statistically significant and the results in the class diagram contradict those in the state machine. We believe that these results can be accounted by the way we presented footprinting to the participants: a static metamodel footprint in text form. Participants may have overlooked it or may not have used it properly. We would have obtained different results if we had trained the participants to footprinting or if we had provided them with a modeling tool displaying feedback on the confinement and the completeness of their models.

Further research is needed for establishing the impact of footprinting on model quality. Furthermore, footprinting is not only meant for creating better models, it can also be used to better understand model operations. Thus, investigating the impact of footprinting on the comprehension of an operation is another direction for demonstrating the benefits of footprinting empirically. Finally, one could evaluate to what extent a set of operations is a good characterization of a model's purpose.

Chapter 6

Conclusions

6.1 Summary and Achievements

Software engineering involves many models to document and analyze software systems. The strength of modeling stems from the ideas of abstraction and separation of concern. Thus, models loose much of their value if they are not at the right level of abstraction for their purpose. This motivates us to investigate objective measurement of a model's abstractness and systematic guidance to attain the right level of abstraction with respect to a given purpose. To reason about the purpose of a model, we propose to characterize the purpose with the set of operations that the model enables:

Thesis Statement. *The purpose of a model can be characterized by the set of operations that this model enables.*

We summarize our work by answering the research questions presented in Section 1.3.

Question 1. *How can we capture the purpose(s) of a model?*

In Chapter 2, we present two approaches based on existing goal-oriented techniques: Goal-Operation-Metamodel (GOM) and Intentional Metamodeling (IM). GOM extends the Goal-Question-Metrics paradigm to support models other than measurement. In this approach, modeling goals are stated informally at the conceptual level. Goals are then refined at the operational level into model operations and, at the definable level into a metamodel. IM uses KAOS models as metamodels. KAOS is a modeling language for early requirements phases. An intentional metamodel contains 4 views: the goal, the responsibility, the operation and the object view. In both approaches, the modeling goals are eventually operationalized with a set of model operations and a metamodel supporting them.

Question 2. *How can we measure and improve the confinement of a model with respect to a given purpose?*

In Chapter 3, we introduce the concept of model footprint. The footprint of an operation is the set of model elements used by this operation during its execution. When a model enables multiple operations, the footprint is the combination of the footprints left by each individual operation. Modelers can identify confinement issues in a model by analyzing the extent to which it is covered by the footprint left by the set of operations characterizing its purpose.

We presented two techniques to compute footprints: dynamic and static footprinting. Dynamic footprinting reveals the actual footprint of an operation, but requires its execution. In contrast, static footprinting estimates the footprint of an operation without executing it by analyzing its source code. During this analysis, we compute the metamodel footprint of the operation, that is, the set of all constructs involved in the definition of the operation. The model footprint can then be obtained by selecting from the model only those elements that are instances of the constructs in the metamodel footprint. Our experiments suggest that static footprinting can estimate actual footprints with a high precision while being many times faster than dynamic footprinting.

Question 3. *How can we measure and improve the completeness of a model with respect to a given purpose?*

Chapter 4 focuses on metamodel footprints, which are computed during static footprinting. Completeness issues can be detected by comparing the metamodel footprint with the expected metamodel footprint, which is the part of the metamodel covered by the model.

Question 4. *What is the impact of our approach on model quality?*

Chapter 5 reports on a pair of controlled experiments involving students from Rennes and Zurich. In these experiments, students had to accomplish two modeling tasks: In one of them, they were

given the metamodel footprint, in the other not. The studies failed to demonstrate any benefits of our approach on model quality. However, this result may have many explanations, all related to our experimental design: the participants were not trained for the approach, we did not provide them with proper tool support and the case studies used in the modeling tasks were too simple. Thus, the studies are inconclusive: The experiments do not support our thesis statement, but they do not invalidate it either.

In Chapter 4, we demonstrate that static footprinting can be used to validate model operations. In a case study, four MDE experts were able to detect some bugs in model operations with the help of footprints without looking at the source code of the operations.

While these results are insufficient to claim that we achieved our goal, we nevertheless made some significant steps towards it. The main components of MiRiA are valid contributions to the field of software engineering:

1. Two approaches — Goal-Operation-Metamodel and Intentional Metamodeling — for capturing the purposes of models and operationalizing them with a set of model operations.
2. A method based on footprinting for improving the confinement and completeness of models with respect to a set of model operations.
3. A method for validating model operations using metamodel footprints.

4. Two techniques — dynamic and static footprinting — to calculate the footprint of a model operation.

6.2 Limitations and Future Work

MiRiA is one of the first attempt towards an objective measure of the level of abstraction of models. In this section, we discuss some limitations of our work and suggest possible directions for future work.

Footprinting identifies elements that have not been used by a set of model operations or suggests elements that may have been used by these operations if they existed. As such, it can act as a heuristic to assess and improve a model's level of abstraction given the set of operation that this model enables. However, there are many explanations for the presence of non-used elements in a model: the model may be at the right level of abstraction but it has the wrong scope, these elements are comments to improve the comprehensibility of the model or the operations are wrong or immature. In other words, the footprinting method may reveal problems that have nothing to do with models being at the wrong level of abstraction. This issue may be addressed by further distinguishing the various roles of model elements such as comment, containment or navigation.

Static footprints estimate dynamic footprints. In most cases, the precision of static footprints is good enough to validate models and

operations. However, two issues may be problematic. First, static footprinting ignores conditions on the state of objects. Second, if a feature is defined in a metaclass that has many subclasses, then the corresponding settings will be selected in all instances of this metaclass. Both issues are illustrated in the state machine example of Chapter 5. In that case study, some pseudo-states (those of the initial kind) are relevant, but not the others (*e.g.*, those of the junction kind). Furthermore, the name of normal states is relevant, but not the name of final states. These issues can be addressed by annotating the metamodel footprint with constraints that must be satisfied by an element in order to be included in the static footprint. However, inferring these constraints automatically from the source code of an operation remains an open problem.

MiRiA advocates eliciting and documenting the purpose of a model. This may be a very difficult exercise for two reasons. First, modelers are not experienced with intentional modeling, not even considering intentional metamodeling. Indeed, courses in Software Engineering typically cover data, behavior and process modeling languages but leave out goal modeling. Second, the purpose of a model is not always clear and carved in stone. Most likely, engineers will have to follow a process similar to the iterative requirements process for discovering requirements [RR13]. In our work, we only made the first steps towards intentional metamodeling by adapting two existing goal-oriented techniques. Templates, guidelines and processes still

need to be elaborated to support engineers in intentional metamodeling.

Our empirical evaluation of footprinting failed to demonstrate its benefits on model quality. As we explained in Chapter 5, this result can be explained by a faulty experiment design. Thus, it may be worth to reconduct this experiment with the following changes: train the participants, use a more complicated case study and provide proper tool support for footprinting.

Tool support is especially important for quality assessment because good metrics are — among other things — reliable and cheap to compute. The reliability of a metric ensures that its results are reproducible, while high costs discourage people from using the metrics. During the interviews presented in Chapter 4, experts highlighted the importance of proper tooling. Besides, the inconclusive results from the experiment reported in Chapter 5 may be due to the lack of adequate tool support. So far, we have implemented static and dynamic footprinting for operations written in Kermeta [JGB11b]. Still, much work remains to be done to implement a proper toolset for the MiRiA approach. Among others, such a toolset should include (1) an editor to visualize footprints and to specify footprints in a consistent manner, (2) an automatic comparison between the actual and the expected footprints, (3) metrics for confinement and completeness and (4) an integration in popular modeling environments. Such as toolset would provide engineers with immediate and usable feedback on the confinement and completeness of their models.

Despite these limitations, we believe that MiRiA will help modelers in modeling at the right level of abstraction and we hope that it will inspire future research in this direction.

Bibliography

[Amb05] Scott W. Ambler. *The Elements of UML (TM) 2.0 Style*. Cambridge University Press, New York, NY, USA, 2005.

[AS08] Colin Atkinson and Dietmar Stoll. Orthographic modeling environment. In *Fundamental Approaches to Software Engineering (FASE 2008)*, volume 4961 of *Lecture Notes in Computer Science*, pages 93–96, Budapest, Hungary, 2008. Springer.

[Bas92] Victor R. Basili. Software modeling and measurement: the goal/question/metric paradigm. Technical Report UMIACS TR-92-96, University of Maryland, 1992.

[BB06] Brian Berenbach and Gail Borotto. Metrics for model driven requirements development. In *28th International Conference on Software Engineering (ICSE '06)*, pages 445–451, Shanghai, China, 2006. ACM.

[BCBB12] Arnaud Blouin, Benoit Combemale, Benoit Baudry, and Olivier Beaudoux. Kompren: Modeling and generating model slicers. *Software and Systems Modeling*, pages 1–17, 2012.

[Ber04] Brian Berenbach. The evaluation of large, complex UML analysis and design models. In *26th International Conference on Software Engineering (ICSE '04)*, pages 232–241, Edinburgh, Scotland, UK, 2004. IEEE Computer Society.

[Béz05] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[BHRV08] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In *4th International Conference on Graph Transformations (ICGT '08)*, volume 5214 of *Lecture Notes in Computer Science*, pages 396–410, Leicester, UK, 2008. Springer.

[BLYDP04] Lionel C. Briand, Yvan Labiche, Han Daphne Yan, and Massimiliano Di Penta. A controlled experiment on the impact of the object constraint language in UML-based development. In *20th International Conference on Software Maintenance (ICSM 2004)*, pages 380–389, Chicago, IL, USA, 2004. IEEE Computer Society.

[CGB+02] Paul C. Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2002.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[CKM02] Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: the tropos project. *Information Systems*, 27(6):365–389, 2002.

[CM00] Vittorio Cortellessa and Raffaela Mirandola. Deriving a queueing network based performance model from UML diagrams. In *2nd International Workshop on Software and Performance (WOSP 00)*, pages 58–70, Ottawa, Canada, 2000. ACM.

[CRMG08] Leslie Cheung, Roshanak Roshandel, Nenad Medvidovic, and Leana Golubchik. Early prediction of software component reliability. In *30th International Conference on Software Engineering (ICSE '08)*, pages 111–120, Leipzig, Germany, 2008. ACM.

[DOJ+93] Alan Davis, Scott Overmyer, Kathleen Jordan, Joseph Caruso, Fatma Dandashi, Anhtuan Dinh, Gary Kincaid, Glen Ledeboer, Patricia Reynolds, Pradip Sitaram,

Anh Ta, and Mary Theofanos. Identifying and measuring quality in a software requirements specification. In *First International Software Metrics Symposium (METRICS 1993)*, pages 141–152, Baltimore, MD, USA, May 1993. IEEE Computer Society.

[ECFGP09] Sergio España, Nelly Condori-Fernandez, Arturo Gonzalez, and Óscar Pastor. Evaluating the completeness and granularity of functional requirements specifications: A controlled experiment. In *17th International Conference on Requirements Engineering (RE '09)*, pages 161–170, Atlanta, GA, USA, 2009. IEEE Computer Society.

[Egy06] Alexander Egyed. Instant consistency checking for the UML. In *28th International Conference on Software Engineering (ICSE '06)*, pages 381–390, Shanghai, China, 2006. ACM.

[Egy07] Alexander Egyed. Fixing inconsistencies in UML design models. In *29th International Conference on Software Engineering (ICSE '07)*, pages 292–301, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[Ern03] Michael D. Ernst. Static and dynamic analysis: synergy and duality. In *Workshop on Dynamic Analysis (WODA 2003)*, pages 24–27, Portland, OR, USA, 2003. IEEE Computer Society.

[ESSD08] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. *Selecting Empirical Methods for Software Engineering Research*, chapter 11, pages 285–311. Springer, 2008.

[FGDTS06] Robert B. France, Sudipto Ghosh, Trung T. Dinh-Trong, and Arnor Solberg. Model-driven development using UML 2.0: Promises and pitfalls. *IEEE Computer*, 39(2):59–66, 2006.

[FR07] Robert B. France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Workshop on the Future of Software Engineering (FOSE '07)*, pages 37–54, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[GBJ02] Martin Glinz, Stefan Berner, and Stefan Joos. Object-oriented modeling with ADORA. *Information Systems*, 27(6):425–444, 2002.

[GJM02] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[GR04] Peter Green and Michael Rosemann. Applying ontologies to business and systems modeling techniques and perspectives: Lessons learned. *Journal of Database Management*, 15(2):105–117, 2004.

[GW05] Tony Gorschek and Claes Wohlin. Requirements abstraction model. *Requirements Engineering*, 11(1):79–101, 2005.

[HPvdW05] Stijn Hoppenbrouwers, Henderik A. Proper, and Theo van der Weide. A fundamental view on the process of conceptual modeling. In *24th International Conference on Conceptual Modeling (ER '05)*, volume 3716 of *Lecture Notes in Computer Science*, pages 128–143, Klagenfurt, Austria, 2005. Springer.

[HR04] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[HRK11] Markus Herrmannsdoerfer, Daniel Ratiu, and Maximilian Koegel. Metamodel usage analysis for identifying metamodel improvements. In *3rd International Conference on Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 62–81, Eindhoven, The Netherlands, 2011. Springer.

[IEE00] IEEE. Recommended practice for architectural description of software-intensive systems. *IEEE Std 1471-2000*, 2000.

[Jea08] Cédric Jeanneret. An analysis of model composition approaches. Master's thesis, EPFL, 2008.

[JGB11a] Cédric Jeanneret, Martin Glinz, and Benoit Baudry. Estimating footprints of model operations. In *33rd International Conference on Software Engineering (ICSE 2011)*, pages 601–610, Waikiki, Honolulu, HI, USA, 2011. ACM.

[JGB11b] Cédric Jeanneret, Martin Glinz, and Benoit Baudry. Footprinting operations written in Kermeta. Technical Report IFI-2011.0002, University of Zurich, 2011.

[JGB12] Cédric Jeanneret, Martin Glinz, and Thomas Baar. Modeling the purposes of models. In *Modellierung 2012*, volume 201 of *Lecture Notes in Informatics*, pages 11–26, Bamberg, Germany, 2012. GI.

[JGBC12] Cédric Jeanneret, Martin Glinz, Benoit Baudry, and Benoit Combemale. Impact of footprinting on model quality: An experimental evaluation. In *2nd International Workshop on Model-Driven Requirements Engineering (MoDRE 2012)*, pages 77–86, Chicago, IL, USA, 2012.

[KAER06] Jochen Küster and Mohamed Abd-El-Razik. Validation of model transformations — first experiences using a white box approach. In *3rd International Workshop on Model Development, Validation and Verification (MoDeVa '06)*, volume 4364 of *Lecture Notes in Computer Science*, pages 193–204, Genoa, Italy, 2006. Springer.

[Kle08] Anneke G. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.

[Kra07] Jeff Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, 2007.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[KSJ06] John Krogstie, Guttorm Sindre, and Havard Jorgensen. Process models representing knowledge for action: a revised quality framework. *European Journal of Information Systems*, 15(1):91–102, February 2006.

[KSTV03] Bogdan Korel, Inderdeep Singh, Luay Tahat, and Boris Vaysburg. Slicing of state-based models. In *19th International Conference on Software Maintenance (ICSM 2003)*, pages 34–43, Amsterdam, The Netherlands, 2003. IEEE Computer Society.

[Küh06] Thomas Kühne. Matters of (meta-) modeling. *Software and Systems Modeling*, 5(4):369–385, December 2006.

[LG00] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[LSS94] Odd Ivar Lindland, Guttorm Sindre, and Arne Sølvberg. Understanding quality in conceptual modeling. *IEEE Software*, 11(2):42–49, 1994.

[Lud03] Jochen Ludewig. Models in software engineering - an introduction. *Software and Systems Modeling*, 2(1):5–14, March 2003.

[LZ74] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, Santa Monica, California, United States, 1974. ACM.

[MA07] Parastoo Mohagheghi and Jan Aagedal. Evaluating quality in model-driven engineering. In *International Workshop on Modeling in Software Engineering (MISE '07)*, page 6, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[MDN09] Parastoo Mohagheghi, Vegard Dehlen, and Tor Neple. Definitions and approaches to model quality in model-based software development: A review of literature. *Information and Software Technology*, 51(12):1646–1669, 2009.

[MFBC12] Pierre-Alain Muller, Frédéric Fondement, Benoit Baudry, and Benoît Combemale. Modeling modeling modeling. *Software and Systems Modeling*, 11(3):347–359, 2012.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005)*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278, Montego Bay, Jamaica, 2005. Springer.

[MMP88] Ole Madsen and Birger Møller-Pedersen. What object-oriented programming may be - and what it does not have to be. In *European Conference on Object-Oriented Programming (ECOOP '88)*, volume 1445 of *Lecture Notes in Computer Science*, pages 1–20, Brussels, Belgium, 1988. Springer.

[Moo05] Daniel L. Moody. Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. *Data & Knowledge Engineering*, 55(3):243–276, 2005.

[MSBS03] Daniel L. Moody, Guttorm Sindre, Terje Brasethvik, and Arne Sølvberg. Evaluating the quality of information models: Empirical testing of a conceptual model

quality framework. In *25th International Conference on Software Engineering (ICSE '03)*, pages 295–305, Portland, OR, USA, 2003. IEEE Computer Society.

[NKF93] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. Expressing the relationships between multiple views in requirements specification. In *15th International conference on Software Engineering (ICSE '93)*, pages 187–196, Baltimore, MD, USA, 1993. IEEE Computer Society.

[NKF94] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, 1994.

[Nug09] Ariadi Nugroho. Level of detail in UML models and its impact on model comprehension: A controlled experiment. *Information and Software Technology*, 51(12):1670–1685, 2009.

[OMG03] OMG. MDA guide version 1.0.1, omg/03-06-01, 2003.

[OMG11a] OMG. Meta Object Facility version 2.4.1, formal/2011-08-07, 2011.

[OMG11b] OMG. Query/View/Transformation specification version 1.1, formal/2011-01-01, 2011.

[OMG11c] OMG. UML superstructure specification version 2.4.1, formal/2011-08-06, 2011.

[OMG12] OMG. Object Constraint Language specification version 2.3.1, formal/2012-01-01, 2012.

[Rob11] Colin Robson. *Real World Research: A Resource for Users of Social Research Methods in Applied Settings*. John Wiley & Sons Ltd, 3rd edition, 2011.

[Rot89] Jeff Rothenberg. The nature of modeling. In *Artificial intelligence, simulation & modeling*, pages 75–92. John Wiley & Sons, Inc., New York, NY, USA, 1989.

[RR13] Suzanne Robertson and James Robertson. *Mastering the Requirements Process: Getting Requirements Right*. Addison-Wesley Professional, 3 edition, 2013.

[RW05] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005.

[SAA+02] Dag I. K. Sjøberg, Bente Anda, Erik Arisholm, Tore Dybå, Magne Jørgensen, Amela Karahasanovic, Espen F. Koren, and Marek Vokác. Conducting realistic

experiments in software engineering. In *1st International Symposium on Empirical Software Engineering (ISESE 2002)*, pages 17–26, Nara, Japan, 2002. ACM.

[Sal10] Hanania T. Salzer. Abstraction level hierarchy: The model and its significance for software engineering. In *International Conference on Software Science, Technology & Engineering (SWSTE '10)*, pages 61–69, Herzlia, Israel, 2010. IEEE Computer Society.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In *20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '94)*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163, Herrsching, Germany, 1994. Springer.

[Sei03] Ed Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, 2003.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[SJB08] John W. Satzinger, Robert B. Jackson, and Stephen D. Burd. *Systems Analysis and Design in a Changing World*. International Thomson Computer Press, Boston, MA, USA, 2008.

[SMBJ09] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel. Meta-model pruning. In *12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009)*, volume 5795 of *Lecture Notes in Computer Science*, pages 32–46, Denver, CO, USA, 2009. Springer.

[SMM+12] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry, and Jean-Marc Jézéquel. Reusable model transformations. *Software and Systems Modeling*, 11(1):111–125, 2012.

[SR98] Reinhard Schuette and Thomas Rotthowe. The guidelines of modeling: An approach to enhance the quality in information models. In *17th International Conference on Conceptual Modeling (ER '98)*, volume 1507 of *Lecture Notes in Computer Science*, pages 240–254, Singapore, 1998. Springer.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[SZ07] Lorenza Saitta and Jean-Daniel Zucker. Abstraction and complexity measures. In *7th Symposium on Abstraction, Reformulation, and Approximation (SARA 2007)*, volume 4612 of *Lecture Notes in Computer Science*, pages 375–390, Whistler, Canada, 2007. Springer.

[vG91] John P. van Gigch. *System Design Modeling and Metamodeling*. Plenum Press, New York, NY, USA, 1991.

[vL09] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[VVP02] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, 2002.

[WH06] Roel Wieringa and Hans Heerkens. The methodological soundness of requirements engineering papers: a conceptual framework and two case studies. *Requirements Engineering*, 11(4):295–307, September 2006.

[Wir83] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 26(1):70–74, 1983.

[WKC06] Junhua Wang, Soon-Kyeong Kim, and David Carrington. Verifying metamodel coverage of model transformations. In *17th Australian Software Engineering Conference (ASWEC 2006)*, pages 270–282, Sydney, Australia, 2006. IEEE Computer Society.

[WRH+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[WW93] Yair Wand and Ron Weber. On the ontological expressiveness of information systems analysis and design grammars. *Information Systems Journal*, 3(4):217–237, 1993.

[Yu97] Eric Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *3rd International Symposim on Requirements Engineering (RE '97)*, pages 226–235, Annapolis, MD, USA, 1997. IEEE Computer Society.

Appendix A

Modeling Assignments

This appendix presents the modeling assignments that we used in the controlled experiments to evaluate the impact of footprinting on model quality (Chapter 5).

A.1 Dental Clinic

Instructions

The following section describes a system to manage patient records in a dental clinic. Your task is to produce an analysis model of this system with a class diagram based on the description. Your model will be used to generate the skeleton of a glossary. We will compare your model to a reference model: Make sure to include every relevant piece of information, but not more!

Static Metamodel Footprint

Note: The static metamodel footprint has not been handed to all participants.

In this model, we are only interested in the following items:

- Classes
  - name of the class
- Attributes
  - name of the attribute
- Associations
- Association Ends
  - multiplicity of the association end
- Generalizations

Case Description

A clinic with three dentists and several dental hygienists needed a system to help administer patient records. This system does not keep any medical records. It only processes patient administration.

Each patient has a record with his/her name, date of birth, gender, date of first visit, and date of last visit. Patient records are grouped together under a household. A household has attributes such as name of head of household, address, and telephone number. Each

household is also associated with an insurance carrier record. The insurance carrier record contains name of insurance company, address, billing contact person, and telephone number.

In the clinic, each dental staff person also has a record that tracks who works with a patient (dentist, dental hygienist, x-ray technician). Because the system focuses on patient administration records, only minimal information is kept about each dental staff person, such as name, address, and telephone number. Information is maintained about each office visit, such as date, insurance copay amount (amount paid by the patient), paid code, and amount actually paid. Each visit is for a single patient, but, of course, a patient will have many office visits in the system. During each visit, more than one dental staff person may be involved in the visit by doing a procedure. For example, the x-ray technician, dentist, and dental hygienist may all be involved in a single visit. In fact, some dentists are specialists in such things as crown work, and even multiple dentists may be involved with a patient. Detailed information is kept about procedures performed by a staff person during a visit. This information includes type of procedure, description, tooth involved, the copay amount, the total charge, the amount paid, and the amount insurance denied.

Finally, the system also keeps track of invoices. There are two types of invoices: invoices to insurance companies and invoices to heads of household. Both types of invoices are fairly similar, listing each visit, the procedures involved, the patient copay amount, and the total due. Obviously, the totals for the insurance company are different

from the patient amounts owed. Even though an invoice is a report (printed out), it also maintains some information such as date sent, total amount, amount already paid, amount due and also the total received, date received, and total denied. (Insurance companies do not always pay all they are billed.)

Reference Model

A reference model is given in Figure A.1.

A.2 Shipment

Instructions

The following section describes the behavior of a shipment by Union Parcel Shipments. Your task is to document this behavior with a state machine. Your model will be used to generate an implementation based on the *State* design pattern. We will compare your model to a reference model: Make sure to include every relevant piece of information, but not more!

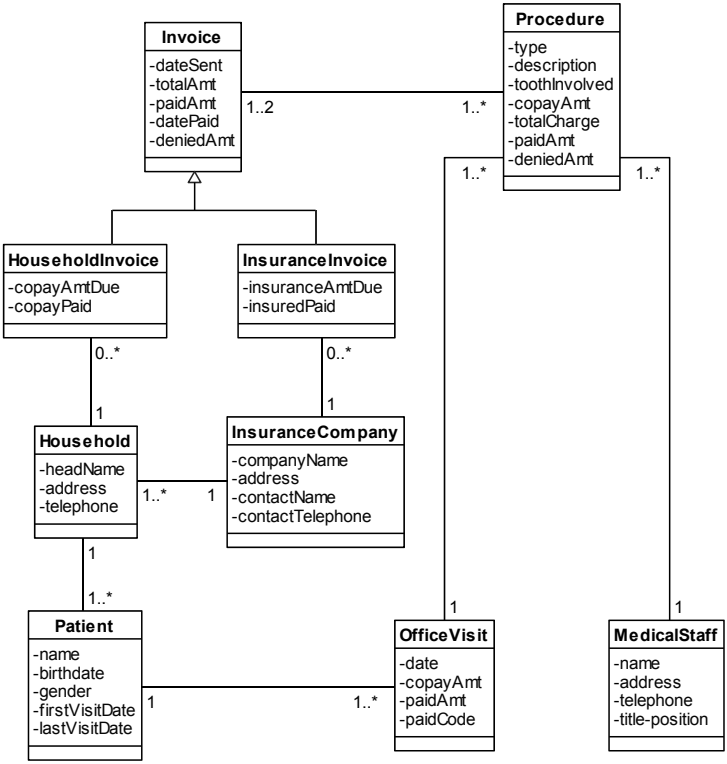


Figure A.1: Reference model for the dental clinic domain.

Static Metamodel Footprint

Note: The static metamodel footprint has not been handed to all participants.

In this model, we are only interested in the following items:

- States
  - name of the state
- Transitions
  - trigger of the transition (as signal event)
- Signal Events
  - name of the event
- Initial States
- Final States

Case Description

A shipment is first recognized after it has been picked up from a customer. After it is in the system, it is considered to be active and in transit. Every time it goes through a checkpoint, such as arrival at an intermediate destination, it is scanned, and a record is created indicating the time and place of the checkpoint scan. The status changes when it is placed on the delivery truck. It is still active, but now it is also considered to have a status of delivery pending. Of course, after it is delivered, the status changes again.

From time to time, a shipment has a destination that is outside the area served by Union. In those cases, Union has working relationships with other courier services. After a package is handed off to another courier, it is noted as being handed over. In those instances, a tracking number for the new courier is recorded (if it is provided). Union also asks the new courier to provide a status change notice after the package has been delivered.

Unfortunately, from time to time a package gets lost. In that case, it remains in an active state for two weeks but is also marked as misplaced. If after two weeks the package has not been found, it is considered lost. At that point, the customer can initiate lost procedures to recover any damages.

Reference Model

A reference model is given in Figure A.2.

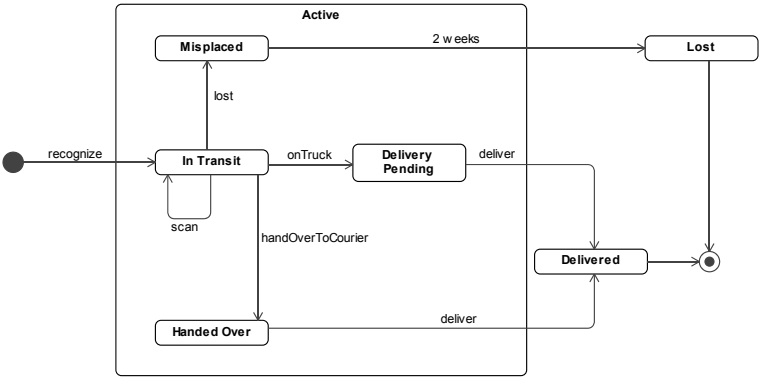


Figure A.2: Reference model for the behavior of a shipment.

Appendix B

Publications

This appendix presents the list of publications on which this cumulative dissertation is built.

B.1 Journal Articles

[JGBC13] Cédric Jeanneret, Martin Glinz, Benoit Baudry, and Benoit Combemale. Analyzing the quality of models and model operations with metamodel footprints. *Software and Systems Modeling*, 2013. Under Review.

B.2 Conference Papers

[JGB11] Cédric Jeanneret, Martin Glinz, and Benoit Baudry. Estimating footprints of model operations. In *33rd International Conference on Software Engineering (ICSE 2011)*, pages 601–610, Waikiki, Honolulu, HI, USA, 2011. ACM.

[JGB12] Cédric Jeanneret, Martin Glinz, and Thomas Baar. Modeling the purposes of models. In *Modellierung 2012*, volume 201 of *Lecture Notes in Informatics*, pages 11–26, Bamberg, Germany, 2012. GI.

B.3 Workshop Papers

[JGBC12] Cédric Jeanneret, Martin Glinz, Benoit Baudry, and Benoit Combemale. Impact of footprinting on model quality: An experimental evaluation. In *2nd International Workshop on Model-Driven Requirements Engineering (MoDRE 2012)*, pages 77–86, Chicago, IL, USA, 2012.

B.4 PhD Symposium

[Jea09] Cédric Jeanneret. Finding the right level of abstraction. In *Doctoral Symposium of the 17th International Requirements Engineering Conference (RE '09)*, Atlanta, GA, USA, 2009.

B.5 Technical Reports

[JGB11] Cédric Jeanneret, Martin Glinz, and Benoit Baudry. Footprinting operations written in Kermeta. Technical Report IFI-2011.0002, University of Zurich, 2011.

Curriculum Vitae

Name:	Cédric Joël Jeanneret
Date of Birth:	February 25, 1983
Place of Citizenship:	Travers, NE, Switzerland

1998-2002	High school, Collège de Sainte-Croix *Fribourg, FR, Switzerland*
2002-2005	Bachelor in Computer Science, EPFL *Lausanne, VD, Switzerland*
2005-2008	Master in Computer Science, EPFL *Lausanne, VD, Switzerland*
2007-2008	Visiting Student, CSU *Fort Collins, CO, USA*
2008-2013	Assistant and doctoral student, UZH *Zurich, ZH, Switzerland*